Jülich Supercomputing Centre

JUQUEEN Extreme Scaling Workshop 2016

February 2016

Dirk Brömmel, Wolfgang Frings, Brian J. N. Wylie (Eds.)

Technical Report · FZJ-JSC-IB-2016-01

JÜLICH
FORSCHUNGSZENTRUM

FORSCHUNGSZENTRUM JÜLICH GmbH
Jülich Supercomputing Centre
D-52425 Jülich, Tel. +49 (2461) 61-6402

Technical Report

JUQUEEN Extreme Scaling Workshop 2016

D. Brömmel, W. Frings, B. J. N. Wylie
(Eds.)

FZJ-JSC-IB-2016-01

(last change: 23.03.2016)

Contents

JUQUEEN Extreme Scaling Workshop 2016

Dirk Brömmel, Wolfgang Frings, and Brian J. N. Wylie
Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH

Executive Summary

From 1 to 3 February 2016, Jülich Supercomputing Centre (JSC) organised the latest edition of its series of IBM Blue Gene Extreme Scaling Workshops. This series started with the 2006 "Blue Gene/L Scaling Workshop" [1] using the 8-rack (16 384 cores) *JUBL*, and then moved to *JUGENE* for the 2008 "Blue Gene/P Porting, Tuning & Scaling Workshop" [2] and dedicated "Extreme Scaling Workshops" in 2009 [3], 2010 [4] and 2011 [5]. These latter three workshops attracted 28 teams selected from around the world to investigate scalability on the most massively-parallel supercomputer at the time with its 294 912 cores. 26 of their codes were successfully executed at that scale, three became ACM Gordon Bell prize finalists, and one participant was awarded an ACM/IEEE-CS George Michael Memorial HPC fellowship.

Last year's workshop [6–9] was the first for the *JUQUEEN* Blue Gene/Q [10, 11], and all seven application teams had within 24 hours successfully ran on all 28-racks (458 752 cores capable of running 1 835 008 processes or threads). With their results, five of the codes later joined the list of High-Q Club [12] codes and one existing member improved their scalability.

The High-Q Club is a collection of the highest scaling codes on *JUQUEEN* and as such requires the codes to run on all 28 racks. Codes also have to demonstrate that they profit from each additional rack of *JUQUEEN* in reduced time to solution when strong scaling a fixed problem size or a tolerable increase in runtime when weak scaling progressively larger problems. Furthermore the application configurations should be beyond toy examples and we encourage use of all available hardware threads which is often best achieved via mixed-mode programming. Each code is then individually evaluated based on its weak or strong scaling results with no strict limit on efficiency. Extreme-scaling workshops thus provide an opportunity for additional candidates to prove their scalability and qualify for membership, or – as was the case for some of the codes this and last year – improve on the scaling and efficiency that they had already achieved.

The *MAXI* mini-symposium [13] at the ParCo15 conference enabled five High-Q Club members (including three workshop participants) to present and discuss their experience scaling their applications on a variety of the largest computing systems including *Hermit*, *K computer*, *Mira*, *Piz Daint*, *Sequoia*, *Stampede*, *SuperMUC*, *Tianhe-2* and *Titan*. Exchange of application extreme-scaling experience from these and other leadership HPC computer systems was also the focus of the full-day *aXXLs* workshop at the ISC-HPC conference [14].

Eight international application teams were selected for this year's three day workshop, and given dedicated use of the entire *JUQUEEN* system for a period of over 50 hours. Many of the teams' codes had thematic overlap with JSC Simulation Laboratories or were part of an ongoing collaboration with one of the SimLabs for Fluids & Solids Engineering, Neuroscience, and Terrestrial Systems. While most of the application teams were experienced users of *JUQUEEN* (and similar Blue Gene/Q systems), and had successfully scaled their application codes previously, additional time was scheduled and support from JSC Cross-Sectional Teams was available to do performance analyses and investigate optimisation opportunities.

Particular thanks are due to Jutta Doctor, Jens Henrik Göbbert, Klaus Görgen, Inge Gutheil, Andreas Lintermann, Sebastian Lührs, Alex Peyser, Wendy Sharples, Michael Stephan, Kay Thust and Ilya Zhukov, and the workshop participants themselves who openly shared their own knowledge and expertise.

The eight participating code teams[1] were:

- CIAO *multiphysics, multiscale Navier-Stokes solver for turbulent reacting flows in complex geometries*
  Mathis Bode, **Abhishek Deshmukh** and Heinz Pitsch (RWTH Aachen University, ITV Institute for Combustion Technology)
- Code_Saturne *CFD based on the finite volume method to solve Navier-Stokes equations*
  Charles Moulinec, Vendel Szeremi and David Emerson (STFC Daresbury Laboratory, UK) and Yvan Fournier (EDF R&D, France)
- ICI *simulation based on an implicit finite-element formulation including anisotropic mesh adaptation*
  Hugues Digonnet (Institut de Calcul Intensif, École Centrale de Nantes, France)
- iFETI *implicit solvers for finite-element problems in nonlinear hyperelasticity & plasticity*
  Alex Klawonn, **Martin Lanser** (Universität zu Köln) and **Oliver Rheinbach** (Technische Universität Bergakademie Freiberg)
- NEST-import *module to load neuron and synapse information into the NEST neural simulation tool*
  Till Schumann and **Fabien Delalondre** (Blue Brain Project, Switzerland)
- p4est *library for parallel adaptive mesh refinement and coarsening*
  Carsten Burstedde and **Johannes Holke** (Universität Bonn)
- PFLOTRAN *massively-parallel subsurface flow and reactive transport*
  Hedieh Ebrahimi, Paolo Trinchero, Jorge Molinero (AMPHOS^{21} Consulting S. L., Spain), **Guido Deissmann** and Dirk Bosbach (FZJ IEK6 and JARA-HPC), Glenn Hammond (Sandia National Labs, USA) and Peter Lichtner (OFM Research, USA)
- Seven-League Hydro (SLH) *astrophysical hydrodynamics with focus on stellar evolution*
  Philip Edelmann and Friedrich Röpke (Heidelberger Institut für Theoretische Studien)

Ultimately seven teams had codes successfully run on the full *JUQUEEN* system. Strong scalability demonstrated by Code_Saturne and SLH, both using 4 OpenMP threads for 16 MPI processes on each compute node for a total of 1 835 008 threads, qualify them for High-Q Club membership. Existing members CIAO and iFETI[2] were able to show that they had additional solvers which also scaled acceptably. Furthermore, large-scale in-situ interactive visualisation was demonstrated with a CIAO simulation using 458 752 MPI processes running on 28 racks coupled via JUSITU to VisIt. The two adaptive mesh refinement utilities, ICI and p4est, showed that they could respectively scale to run with 458 752 and 971 504 MPI ranks, but both encountered problems loading large meshes. Parallel file I/O limitations also hindered large-scale executions of PFLOTRAN, however, poor performance of the NEST-import module was tracked down to an internal data-structure mismatch with the HDF5 file objects that prevented use of MPI collective file reading, which when rectified is expected to enable large-scale neuronal network simulations.

[1] with workshop participants marked in bold
[2] part of the FE2TI suite of solvers formerly referred to as ex_nl/FE^2

A short summary of workshop results follows, looking at the employed programming models and languages, code scalability, tools at scale, and parallel file I/O. A list comes next of the 25 High-Q Club member codes at the end of 2015 used for comparison. Detailed reports provided by each of the participating code-teams are found in the following chapter. These present and discuss more execution configurations and results achieved by the application codes during the workshop.

Summary of Results

Characteristics of the eight workshop codes are summarised in Table 1 and discussed in this section, with scaling performance compared in the following section.

Table 1: 2016 Extreme Scaling Workshop code characteristics. Compiler and main programming languages (excluding external libraries), parallelisation including maximal process/thread concurrency (per compute node and overall) and strong and/or weak scaling type, and file I/O implementation. (Supported capabilities unused for scaling runs on *JUQUEEN* in parenthesis.)

	Programming	Parallelisation				
Code	Compiler / Languages	Tasking	Threading	Concurrency	Type	File I/O
CIAO	XL: Ftn	MPI 16		16: 458 752	S	MPI-IO, HDF5
Code_Saturne	XL: C Ftn	MPI 16	OpenMP 4	64: 1 835 008	S	MPI-IO
ICI	XL: C++	MPI 16		16: 458 572	W	MPI-IO
iFETI	XL: C C++	MPI 32		32: 917 072	W	
NEST-import	XL: C++	MPI 1	OpenMP 16	16: 458 752	S W	HDF5 (MPI-IO)
p4est	XL: C	MPI 32		32: 917 504	?	(MPI-IO)
PFLOTRAN	XL: F03	MPI 16		16: 131 072	S	HDF5 (SCORPIO)
SLH	XL: F95	MPI 16	OpenMP 4	64: 1 835 008	S	MPI-IO

Program & execution characteristics Since Blue Gene/Q offers lower-level function calls for some hardware-specific features that are sometimes not available for all programming languages, a starting point is looking at the languages used. The left of Figure 1 shows a Venn set diagram of the programming language(s) used by the High-Q Club codes. It indicates that all three major programming languages are equally popular (without considering lines of code). Of the 8 workshop codes, three are exclusively written in Fortran, two only in C++, one is C, and the other two combine C with C++ or Fortran. Portability is apparently important, as hardware-specific coding extensions are generally avoided. The workshop codes all used IBM's XL compiler suite, whereas various High-Q Club application codes have preferred GCC or Clang compilers which offer support for more recent language standards. Most optimisations employed by the codes are therefore not specific to Blue Gene (or BG/Q) systems, but can also be exploited on other highly-parallel systems.

The four hardware threads per core of the Blue Gene/Q chip in conjunction with the limited amount of compute node memory suggest to make use of multi-threaded programming. It is therefore interesting to see whether this is indeed the preferred programming model and whether the available memory is an issue. The middle of Figure 1 shows a Venn set diagram of the programming models used by High-Q Club codes, and revealing that mixed-mode programming does indeed dominate. Looking at the workshop codes in particular, all eight used MPI, which is almost ubiquitous for portable distributed-memory parallelisation. dynQCD is the only High-Q Club application employing lower-level machine-specific SPI for maximum

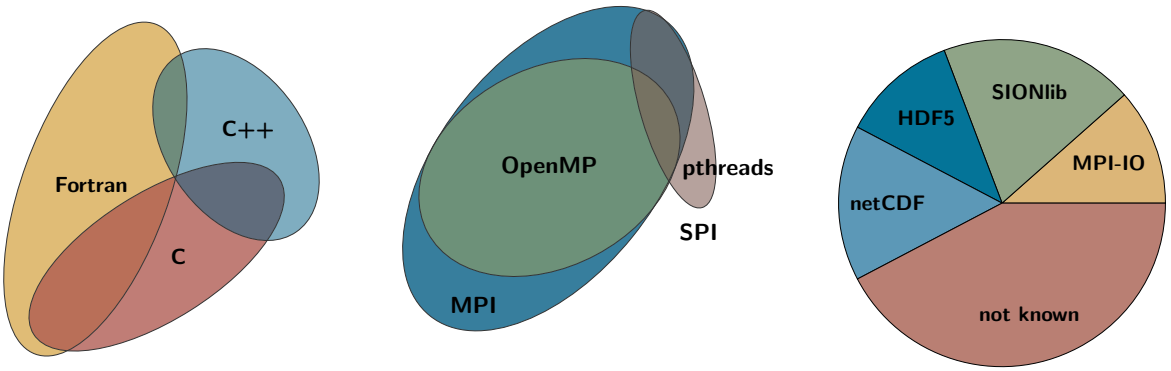


Figure 1: Venn set diagrams of programming languages (left) and parallel programming models (middle), plus a pie-chart showing file I/O (right) used by codes in the High-Q Club.

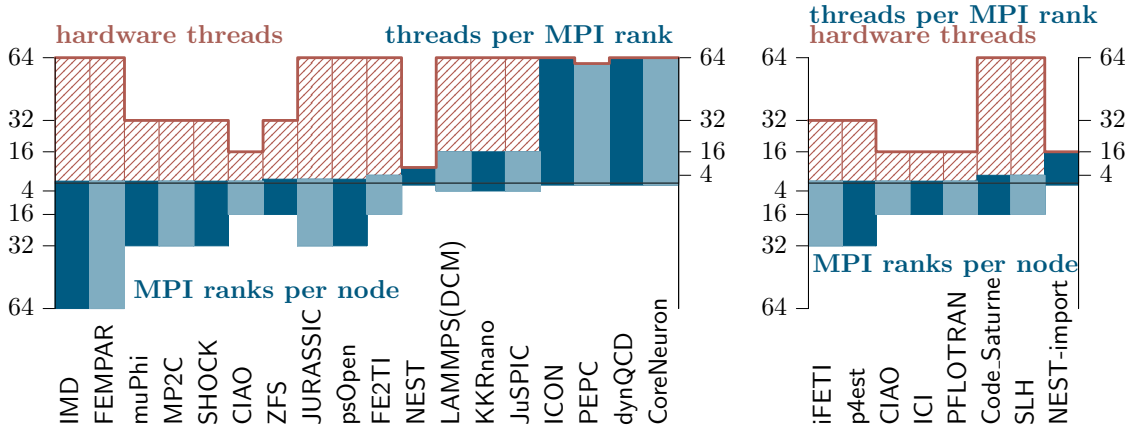


Figure 2: Chart showing the relation between the number of MPI ranks per node and threads per rank used by codes in the High-Q Club (left) and taking part in the workshop (right). The number of resulting hardware threads used on each compute node is shown in red.

performance. Five of the workshop codes exclusively used MPI for their scaling runs, both between and within compute nodes, accommodating to the restricted per-process memory. p4est has started testing the use of MPI-3 shared memory functionality, which is expected to save memory when running multiple MPI processes on each compute node. The remaining three workshop codes employ OpenMP multi-threading to exploit compute node shared memory in conjunction with MPI, as is typical of High-Q Club applications in general. Instead of OpenMP, three of the High-Q Club applications prefer POSIX threading for additional control.

The right of Figure 1 shows a pie-chart breakdown of the I/O libraries used by High-Q Club codes, although in most cases writing output and in some cases reading input files was disabled for their large-scale executions and synthesised or replicated data used instead. Whereas half of the codes in the workshop can use MPI-I/O directly, only 10% of club members can do so. One quarter of the High-Q Club codes can use either (p)HDF5 or (p)NetCDF, despite their often disappointing performance as seen during the workshop. 20% of High-Q Club codes have migrated to using SIONlib for effective parallel I/O, but only a couple of workshop codes have started this. Unfortunately, I/O usage by over 40% of High-Q Club was not provided with their submissions for membership indicating that file I/O has not yet received the required attention.

Figure 2 shows the relation between the number of MPI ranks and threads per compute node where this information was available for High-Q Club (left) and workshop (right) codes. On either side of each diagram are the two extremes of using all 64 hardware threads on each CPU with either 64 processes or 64 threads. Included in red hatching is the resulting number of hardware threads used by the code, i.e., the node concurrency. High-Q Club member codes often seem to benefit from using more hardware threads than physical cores and therefore favour this configuration. For others, such as NEST (and NEST-import), making full use of the available compute node memory for simulations is more important than full exploitation of processor cores and hardware threads. Using lower precision is occassionally exploited to reduce memory requirements and improve time to solution of large-scale simulations, however, larger PFLOTRAN simulations were prohibited by its use of 32-bit (rather than 64-bit) integer indices. The two workshop codes Code_Saturne and SLH which qualified to join the High-Q Club similarly exploit all hardware threads by combining MPI+OpenMP, whereas none of the codes using purely MPI managed to.

Weak and strong scaling and performance Here we show an overview of the scaling results achieved during the workshop as the codes are run on increasing numbers of compute nodes (with more cores and MPI ranks). We compare *strong* (fixed total problem size) and *weak* (fixed problem size per process or thread) scaling, put in context of the scalability results from other codes in the High-Q Club.

Figures 3 and 4 show strong and weak scaling results of the workshop codes, including in grey results from a selection of High-Q Club codes. This indicates the spread in execution results and diverse scaling characteristics of the codes. The graphs show six of the workshop codes managing to run on the full *JUQUEEN* system, and most achieved good scalability. p4est scalability is not included as it had execution times of only a couple of seconds. Note that in many cases the graphs do not have a baseline of a single rack since datasets sometimes did not fit available memory or no data was provided for 1024 compute nodes: for strong scaling an execution with a minimum of seven racks (one quarter of *JUQUEEN*) is accepted for a baseline measurement, with perfect-scaling assumed from a single rack to the baseline.

In Figure 3 almost ideal strong-scaling speed-up of 27× on 28 racks is achieved by CIAO (in both configurations tested, like its previous High-Q Club entry), whereas Code_Saturne only shows a 19× speed-up. SLH speed-up is somewhere in between for the two problem sizes and run configurations measured. dynQCD stands-out with superlinear speed-up of 52×, due to its exceptional ability to exploit caches as problem size per thread decreases, whereas ICON achieved only a modest 12× speed-up.

PFLOTRAN managed to run up to 8 racks before file I/O became prohibitive, but showed reasonable scalability of the solver for sufficiently large problem sizes. NEST-import ran successfully on all 28 racks, but only reached a scalability of 7× (probably largely due to its increasingly inefficient non-collective file reading and all-to-all redistribution).

In Figure 4, the weak scaling efficiency of IciMesh is a respectable 87% with 28 racks (though the largest measurement comes from a somewhat different problem configuration), whereas the new iFETI solver at only 69% scales considerably less well than their current High-Q Club FE2TI solver with 99%. muPhi was able to achieve 102% efficiency on 28 racks compared with a single rack, whereas JURASSIC only managed 68% efficiency due to excessive I/O for the reduced-size test case. Various codes show erratic scaling performance, most likely due to topological effects, e.g. SHOCK is characterised by particularly poor configurations with an odd number of racks in one dimension (i.e. 3, 5 and 7). Similarly, OpenTBL shows marked efficiency drops for non-square numbers of racks (8 and 28).

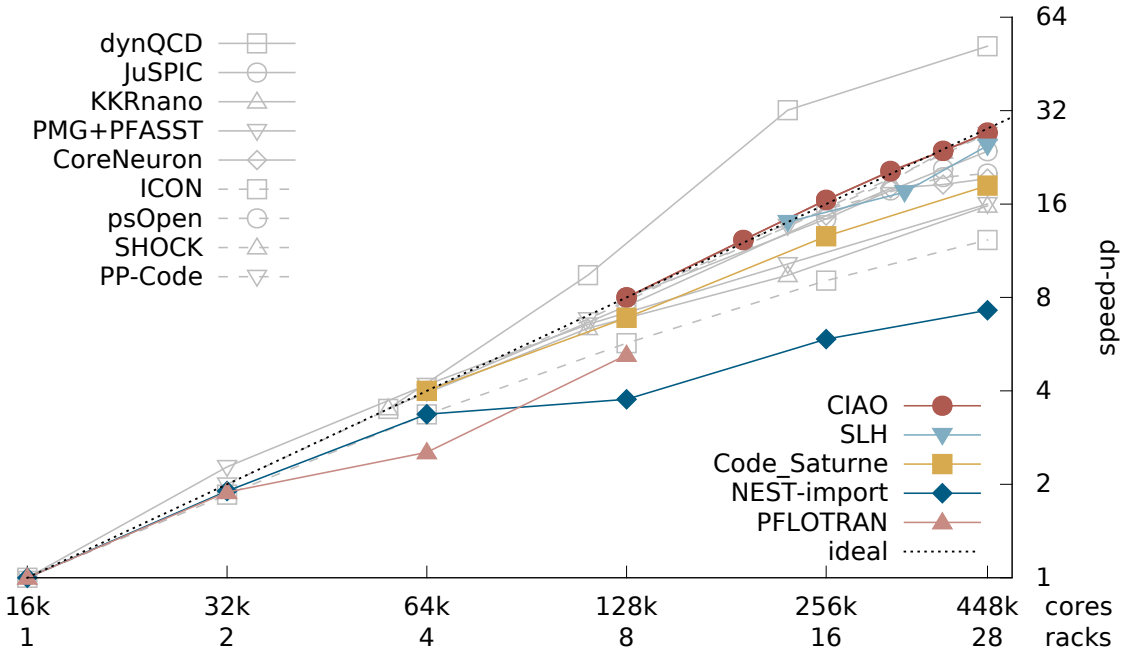


Figure 3: Strong scaling results of the workshop codes with results from existing High-Q Club members included in light grey.

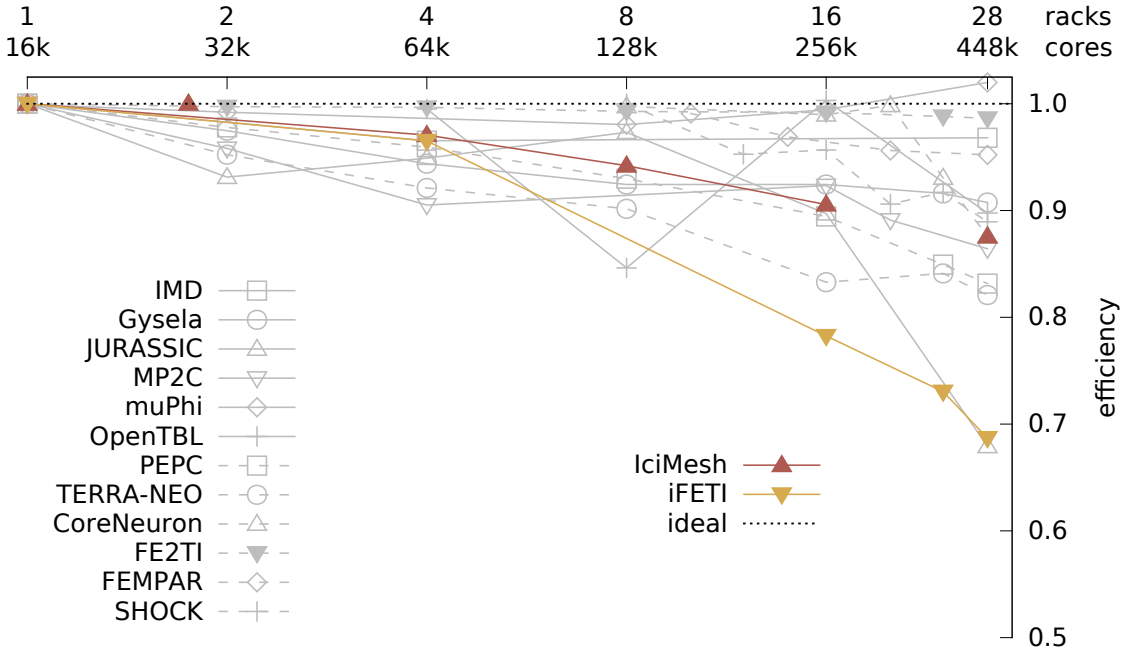


Figure 4: Weak scaling results of the workshop codes with results from existing High-Q Club members included in light grey.

Performance tools Scalasca [15] has been used for performance analysis of applications with over one million MPI processes or threads on *JUQUEEN*. The current version employs the Score-P instrumentation and measurement infrastructure [16], for which execution traces are limited to around 640k processes or threads by the OTF2 trace library. Both profile and trace measurements require dedicated memory for recording measurements during execution and collation of data when measurement concludes. While the amount of memory required increases with scale, unfortunately no report of usage or estimation of requirement is yet provided. Therefore either the full amount of memory not required by the application itself can be specified, or trial-and-error must be employed to determine this.

Scalasca/Score-P was used during the workshop to profile executions of NEST-import on *JUQUEEN*. Since automatic compiler instrumentation of all routines in C++ applications typically results in prohibitive measurement overheads, instead manual annotation of the relevant code regions was used to augment the instrumentation of OpenMP and MPI. An example profile from an execution on all 28-racks of *JUQUEEN* (28 672 MPI ranks each with 16 OpenMP threads, for 458 752 in total) is shown in Figure 5, where large imbalance in MPI file I/O and the main OpenMP parallel region are evident. The former was identified as originating from a mismatch between the module's data structure and the HDF5 file objects which resulted in use of individual MPI file I/O.

While Score-P does not yet support POSIX file I/O or provide measurements of the number of bytes read or written, Darshan [17] was available for this. Figure 6 shows an extract from a Darshan report of a CIAO execution with 65 636 MPI ranks.

Darshan problems with Fortran codes using MPI on *JUQUEEN* were already documented [11], along with the suggested workaround to use `mpif77` when linking (instead of `mpixlf90_r`, etc.). Although this worked for CIAO, it did not for PFLOTRAN, and the C++ codes ICI and iFETI also reported issues (whereas NEST-import was successful). A revised set of Darshan linking wrappers have been developed which are expected to resolve these problems.

Parallel I/O File I/O performance is a critical scalability constraint for many large-scale parallel applications which need to read and write huge datasets or open a large number of files. Half of the workshop codes used MPI file I/O directly, whereas others (e.g. NEST-import) use it indirectly via HDF5. Additionally, p4est can use MPI file I/O but did not for the runs during the workshop.

Code_Saturne used MPI collective file I/O effectively to read 618 GiB of mesh input data, however, writing of simulation output was disabled. The NEST-import module read 1.9 TiB of HDF5 neuron and synapse data but only attained a fraction of the GPFS filesystem bandwidth. Internal data structures are currently being adapted to be able to exploit MPI collective file reading, which is expected to significantly out-perform the current MPI individual/independent file reading and should enable large-scale data-driven neuronal network simulations in future. SLH compared writing 264 GiB of astrophysical simulation output using MPI-IO to a single file or using C stdio to separate files for each MPI process. Writing many process-local files is impractical as it requires all of the files to be created on disk in advance, to avoid filesystem meta-data issues, and an expensive post-processing to aggregate the output into a single file for subsequent use.

While IciMesh was able to generate an adapted mesh with over 100 billion elements using 458 752 MPI ranks on all 28 racks of *JUQUEEN*, the associate solver IciSolve was limited to 65 536 MPI ranks due to problems uncovered with their use of MPI individual/independent file I/O to read their 1.7 TiB mesh files. The parallel adaptive mesh refinement code p4est demonstrated generation, refinement and partitioning, managing in-core meshes with up to 940 billion elements in 21 seconds using 917 504 MPI ranks on 28 *JUQUEEN* racks. In a test

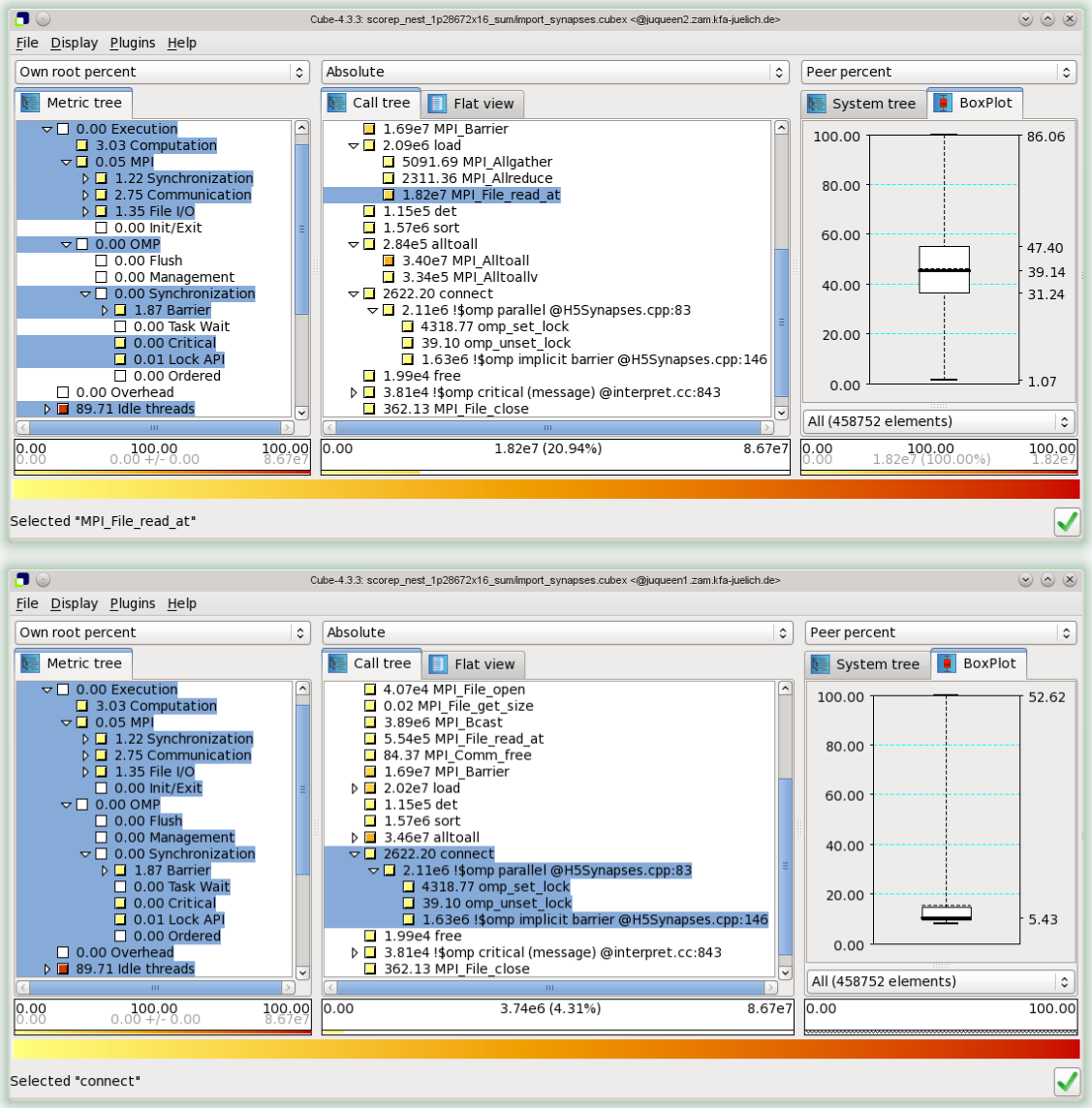
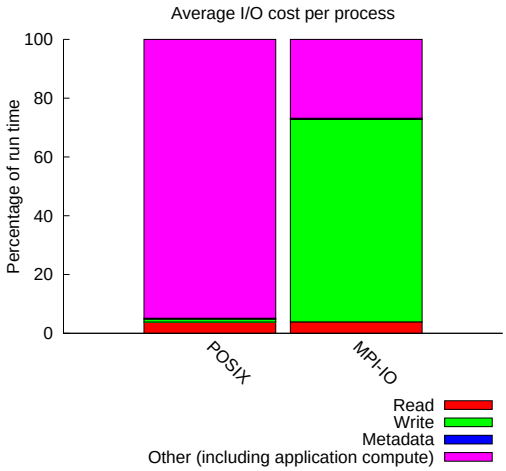
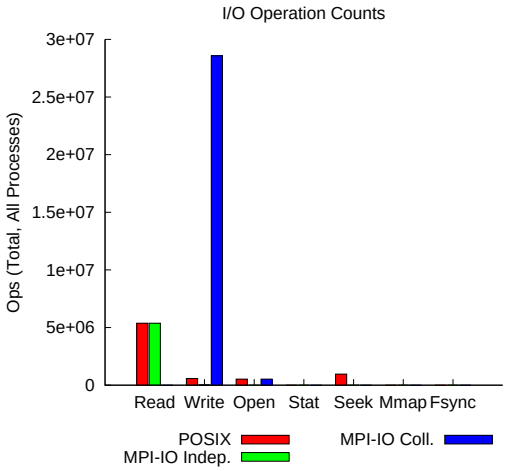
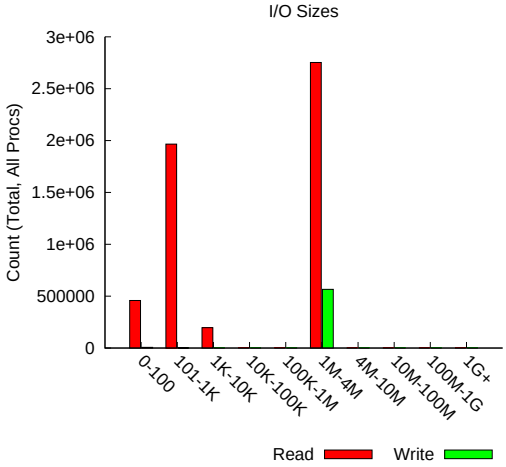


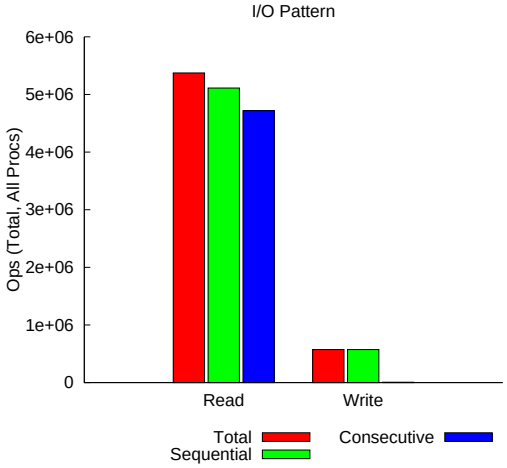
Figure 5: `import_synapses` extract of Scalasca/Score-P profile of NEST-import execution on all 28 racks of *JUQUEEN* (28 672 MPI ranks each with 16 OpenMP threads). The upper view shows time in `MPI_File_read_at` when loading 1.9 TiB of HDF5 neuron and synapse data in six parts (174 calls). Reading time by rank varies from 1.07 to 86.06 seconds, resulting in up to 85 seconds of waiting within the subsequent synchronising collective `MPI_Alltoall`, as part of the 120 seconds each rank takes to load and redistribute the data. Revision of the internal data structure to allow use of the collective `MPI_File_read_at_all` is expected to provide a substantial performance improvement. The lower view shows time in the `connect` step ranging from 4.14 to 52.62 seconds, where some threads wait up to 48 seconds in the implicit barrier closing the OpenMP parallel region.

jobid: 1	uid: 9558	nprocs: 65536	runtime: 973 seconds









Most Common Access Sizes

access size	count
1400000	2818048
512	1769992
4194304	565380
8200	196628

File Count Summary
(estimated by I/O access offsets)

type	number of files	avg. size	max size
total opened	15	148G	553G
read-only files	2	29M	57M
write-only files	13	171G	553G
read/write files	0	0	0
created files	13	171G	553G

./arts_cf <unknown args>

Figure 6: Extract of Darshan report from CIAO *arts_cf* execution with 65536 MPI ranks on four racks of *JUQUEEN*.

case with a larger coarse mesh, memory was the limiting factor for broadcasting data from a single MPI rank to the others. HDF5 file I/O also presented an insurmountable scalability impediment for PFLOTRAN, particularly for larger problem sizes and with more than ten thousand MPI processes.

All of the above is evidence that file I/O is critical and needs the appropriate attention by the programmer and the right methods to perform I/O. At JSC, SIONlib [18] has been developed to address file I/O scalability limitations. It has been used effectively by three High-Q codes (KKRnano, MP2C and muPhi) and several other applications are currently migrating to adopt it (e.g., NEST and SLH).

Apart from specifying a GPFS filesystem type, additional hints for MPI-IO were not investigated by these applications. CIAO experimented with various MPI-IO (ROMIO) hints, but did not observe any benefit when writing single 9 TiB files with MPI file I/O. Whether the parameters have no effect due to the MPI implementation on *JUQUEEN* is still under investigation, but it is well known that reading and writing single shared files fails to exploit the available filesystem bandwidth. An alternative is being able to avoid writing unnecessary data which is one of the motivations for in-situ visualisation with e.g. VisIt. CIAO was able to demonstrate this for a simulation with 458 752 MPI processes on all 28 racks of *JUQUEEN* using the JUSITU coupler.

Miscellany

The workshop participants were all associated with active projects on *JUQUEEN*, which allowed them to prepare their codes and datasets in advance of the workshop. The most successful teams were very familiar with their codes, and how to build and run them in different configurations. Although they were recommended to use the available profiling tools (which some teams had previous experience with) and make available the analysis reports for examination, this advice was ignored such that measurements needed to start from scratch during the workshop itself.

Many of the workshop participants' codes used popular libraries such as HDF5 and PETSc. This facilitated discussion and exchange of experience, despite apparent favouring of different library versions and configurations (which were often customised rather than relying on the system installed versions).

During the workshop, training accounts were provided to participants and access control lists (ACLs) recommended for accessing their individual project accounts. Unfamiliarity with setting up and using ACLs resulted in a variety of problems getting started (including account lock-out).

All of the workshop accounts were part of the training group sharing the provided compute-time allocation and file-system quota. When one team exceeded the 200 TiB group quota, by forgetting to remove test files at the end of their jobs, this temporarily resulted in compilation and execution failures for the other participants. Over the 50 hours of the workshop, 3 500 TiB was read and 124 TiB was written in total between applications and the I/O nodes, with the largest jobs reading 330 TiB and writing 15 TiB respectively. Maximum bandwidths recorded over one minute intervals was 700 GiB/s for reading and 18 GiB/s for writing. Despite the considerable load on the GPFS file-system from large-scale parallel jobs doing file I/O during the workshop, the only issue encountered was variable performance.

LLview was invaluable for monitoring the current use of *JUQUEEN* (Figure 7), additionally showing job energy consumption and file I/O performance, while real-time accounting of jobs during the workshop facilitated tracking of resource usage by each participant. LoadLeveler job scheduling quirks were avoided by deft intervention from sysadmins closely monitoring

Figure 7: LLview monitoring of *JUQUEEN* during workshop during an execution of CIAO on 24 racks (and several smaller jobs), with charts of the CIAO job file I/O bandwidth (middle right), file-system usage of the previous 24 hours (centre), and three-day histories of power usage and job-size mix (centre and upper right).

JUQUEEN during the workshop (day and night). 43 jobs were run on all 28 racks, 15 on 24 racks, 13 on 20 racks, as well as 90 jobs with 16 racks and a total of 15.4 million core-hours.

At the start of the workshop, two defective nodeboards limited initial scaling tests to 24 racks for the first 24 hours, but after their replacement 28-rack jobs were quickly tested by most teams. Over the second night, some large jobs from different participants either performed unexpectedly poorly or failed immediately on start-up, however, upon resubmission they executed and performed as expected. No additional hardware failures were encountered during the workshop (beyond the two defective nodeboards at the very start).

Given the demanding nature of the workshop, considerable flexibility is essential, both from participants and test cases, and regarding scheduling of breaks, sessions and system partitions. Physical presence of at least one member of each code team in the classroom for the workshop is essential for rapid communication. This proved especially the case for a mix of proven highly-scaling codes (ready for the entire compute resource) and the need for smaller-scale tests to investigate problems and verify solutions. The physical configuration of *JUQUEEN* makes the workshop particularly unsuited to small-scale tests with long execution times.

The variety of codes and participants, from different but often related subject areas as well as different institutions and countries, combined with similarly diverse support staff, contributes to the intense yet enjoyable nature of these workshops which have the goal of proving and improving the scalability of applications to be able to exploit current and forthcoming extremely large and complex high-performance computer systems.

High-Q Club codes

The full description of the High-Q Club codes along with developer and contact information can be found on the web page [12]. The current list has 25 codes:

CIAO *multiphysics, multiscale Navier-Stokes solver for turbulent reacting flows in complex geometries*
RWTH-ITV & Sogang University

CoreNeuron *simulation of electrical activity of neuronal networks including morphologically detailed neurons*
EPFL Blue Brain Project

dynQCD *lattice quantum chromodynamics with dynamical fermions*
JSC SimLab Nuclear and Particle Physics & Bergische Universität Wuppertal

FE2TI *scale-bridging incorporating micro-mechanics in macroscopic simulations of multi-phase steels*
Universität zu Köln & TUB Freiberg

FEMPAR *massively-parallel finite-element simulation of multi-physics governed by PDEs*
UPC-CIMNE

Gysela *gyrokinetic semi-Lagrangian code for plasma turbulence simulations*
CEA-IRFM Cadarache

ICON *icosahedral non-hydrostatic atmospheric model*
DKRZ & JSC SimLab Climate Science

IMD *classical molecular dynamics simulations*
Ruhr-Universität Bochum & JSC SimLab Molecular Systems

JURASSIC *solver for infrared radiative transfer in the Earth's atmosphere*
JSC SimLab Climate Science

JuSPIC *fully relativistic particle-in-cell code for plasma physics and laser-plasma interaction*
JSC SimLab Plasma Physics

KKRnano *Korringa-Kohn-Rostoker Green function code for quantum description of nano-materials in all-electron density-functional calculations*
FZJ-IAS

LAMMPS(DCM) *a Dynamic Cutoff Method for the Large-scale Atomic/Molecular Massively Parallel Simulator for classical molecular dynamics simulations*
RWTH-AICES

MP2C *massively-parallel multi-particle collision dynamics for soft matter physics and mesoscopic hydrodynamics*
JSC SimLab Molecular Systems

$\mu\phi$ (muPhi) *modelling and simulation of water flow and solute transport in porous media, algebraic multi-grid solver*
Universität Heidelberg

Musubi *multi-component Lattice Boltzmann solver for flow simulations*
Universität Siegen

NEST *large-scale simulations of biological neuronal networks*
FZJ/INM-6 & IAS-6

OpenTBL *direct numerical simulation of turbulent flows*
Universidad Politécnica de Madrid

PEPC *tree code for N-body simulations, beam-plasma interaction, vortex dynamics, gravitational interaction, molecular dynamics simulations*
JSC SimLab Plasma Physics

PMG+PFASST *space-time parallel solver for systems of ODEs with linear stiff terms*
LBNL, Universität Wuppertal, Università della Svizzera italiana & JSC

PP-Code *simulations of relativistic and non-relativistic astrophysical plasmas*
University of Copenhagen

psOpen *direct numerical simulation of fine-scale turbulence*
RWTH-ITV Inst. for Combustion Technology & JARA

SHOCK *structured high-order finite-difference kernel for compressible flows*
RWTH Shock Wave Laboratory

TERRA-NEO *modelling and simulation of Earth mantle dynamics*
Universität Erlangen-Nürnberg, LMU & TUM

waLBerla *Lattice-Boltzmann method for the simulation of fluid scenarios*
Universität Erlangen-Nürnberg

ZFS *computational fluid dynamics & aero-acoustics, conjugate heat transfer, particulate flows*
RWTH Fluid Mechanics and Inst. of Aerodynamics & JSC SimLab Fluids and Solids Engineering

References

[1] Wolfgang Frings, Marc-André Hermanns, Bernd Mohr & Boris Orth (editors), *Jülich Blue Gene/L Scaling Workshop 2006*, Technical Report FZJ-ZAM-IB-2007-02, Forschungszentrum Jülich, 2007.
`http://juser.fz-juelich.de/record/55967`

[2] Bernd Mohr & Wolfgang Frings, *Jülich Blue Gene/P Porting, Tuning & Scaling Workshop 2008*, Innovatives Supercomputing in Deutschland, inSiDE 6(2), 2008.

[3] Bernd Mohr & Wolfgang Frings (eds), *Jülich Blue Gene/P Extreme Scaling Workshop 2009*, Technical Report FZJ-JSC-IB-2010-02, Forschungszentrum Jülich, Feb. 2010.
`http://juser.fz-juelich.de/record/8924`

[4] Bernd Mohr & Wolfgang Frings (eds), *Jülich Blue Gene/P Extreme Scaling Workshop 2010*, Technical Report FZJ-JSC-IB-2010-03, Forschungszentrum Jülich, May 2010.
`http://juser.fz-juelich.de/record/9600`

[5] Bernd Mohr & Wolfgang Frings (eds), *Jülich Blue Gene/P Extreme Scaling Workshop 2011*, Technical Report FZJ-JSC-IB-2011-02, Forschungszentrum Jülich, Apr. 2011.
`http://juser.fz-juelich.de/record/15866`

[6] Dirk Brömmel, Wolfgang Frings & Brian J. N. Wylie (eds), *JUQUEEN Extreme Scaling Workshop 2015*, Tech. Report FZJ-JSC-IB-2015-01, Forschungszentrum Jülich, Feb. 2015.
`http://juser.fz-juelich.de/record/188191`

[7] Dirk Brömmel, Wolfgang Frings & Brian J. N. Wylie, *High-Q en route to exascale*, Poster at Exascale Applications and Software Conference (EASC, Edinburgh, UK), Apr. 2015.
`http://juser.fz-juelich.de/record/202370`

[8] Dirk Brömmel, Wolfgang Frings & Brian J. N. Wylie, *Extreme-scaling applications 24/7*, Poster at 30th Int'l Conf. ISC High Performance (Frankfurt, Germany), July 2015.
`http://juser.fz-juelich.de/record/202371`

[9] Dirk Brömmel, Wolfgang Frings & Brian J. N. Wylie, *Extreme-scaling applications 24/7 on JUQUEEN Blue Gene/Q*, to appear in Proc. Int'l Conf. on Parallel Computing (ParCo, Edinburgh, UK), Sept. 2015. http://juser.fz-juelich.de/record/276404

[10] Michael Stephan & Jutta Doctor, *JUQUEEN: IBM Blue Gene/Q supercomputer system at the Jülich Supercomputing Centre*, Journal of Large-Scale Research Facilities **1** A1 (2015). http://dx.doi.org/10.17815/jlsrf-1-18

[11] JUQUEEN: Jülich Blue Gene/Q. http://www.fz-juelich.de/ias/jsc/juqueen

[12] The High-Q Club at JSC. http://www.fz-juelich.de/ias/jsc/high-q-club

[13] Dirk Brömmel, Wolfgang Frings & Brian J. N. Wylie, *Multi-system Application Extreme-scaling Imperative*, Mini-symposium at Int'l Conf. on Parallel Computing (ParCo, Edinburgh, UK), Sept. 2015. http://www.fz-juelich.de/ias/jsc/MAXI

[14] Dirk Brömmel, Wolfgang Frings & Brian J. N. Wylie, *Application Extreme-scaling Experience of Leading Supercomputing Centres*, Workshop at 30th Int'l Conf. ISC High Performance (Frankfurt, Germany), July 2015. http://www.fz-juelich.de/ias/jsc/aXXLs

[15] Scalasca: Toolset for scalable performance analysis of large-scale parallel applications. http://www.scalasca.org/

[16] Score-P: Community-developed scalable instrumentation and measurement infrastructure. http://www.score-p.org/

[17] Darshan: HPC I/O characterisation tool, Argonne National Laboratory. http://www.mcs.anl.gov/research/projects/darshan/

[18] SIONlib: Scalable I/O library for parallel access to task-local files. http://www.fz-juelich.de/jsc/sionlib

CIAO: Multiphysics, multiscale Navier-Stokes solver for turbulent reacting flows in complex geometries

Mathis Bode[1], Abhishek Deshmukh[1], Jens Henrik Göbbert[2], and Heinz Pitsch[1]
[1]Institute for Combustion Technology, RWTH Aachen University, Germany
[2]Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Germany

Description of the Code

CIAO (**C**ompressible/**I**ncompressible **A**dvanced reactive turbulent simulations with **O**verset) is a multiphysics, multiscale Navier-Stokes solver for turbulent reacting flows in complex geometries. It performs Direct Numerical Simulations (DNS) as well as Large-Eddy Simulations (LES) based on the Navier-Stokes equations along with multiphysics effects (multiphase, combustion, soot, spark, ...). It is a structured, finite difference code, which enables the coupling of multiple domains and their simultaneous computation. Moving meshes are supported and overset meshes can be used for local mesh refinement. A fully compressible as well as an incompressible/low-Mach solver are available within the code framework. Spatial and temporal staggering of flow variables are used in order to increase the accuracy of stencils. The subfilter model for the momentum equations is an eddy viscosity concept in form of the dynamic Smagorinsky model with Lagrangian averaging along fluid particle trajectories. While the fully compressible solver uses equation of states or tabulated fluid properties, a transport equation for internal/total energy, and a low-storage five-stage explicit Runge-Kutta method for time integration, the incompressible/low-Mach solver uses Crank-Nicolson time advancement and an iterative predictor corrector scheme. The resulting Poisson equation for pressure is solved by HYPRE's multi-grid solver (AMG) or a BiCGStab method. The momentum equations are spatially discretized with central schemes of arbitrary order and various different schemes are available for the scalar equations (WENO, HOUC, QUICK, BQUICK, ...).

The code is written in Fortran and parallelized with MPI. MPI I/O is used for writing the simulation state to the file system. JUSITU [1] and VisIt [2] are used for in-situ visualization. The code has been successfully run on multiple supercomputers (JUQUEEN, SuperMUC, MareNostrum III, ...) and showed good scaling up to 500000 cores.

Figure 1 shows the evolution of a droplet, which is formed from a ligament in a primary breakup simulation performed with CIAO. These simulations are important for optimizing the mixing in engines and reduce the pollutant formation.

Results

This section summarizes the results of the extreme scaling workshop with respect to CIAO. We focused mainly on two different things: First, we studied the performance of CIAO with respect to the different solvers (compressible: *arts_cf* & low Mach: *arts*), different clusters (JSC-JUQUEEN, RWTH-BULLW, ITV-NEHAL, ITV-OXYFLAME), and I/O. Second, we managed to run a simulation on the full JUQUEEN with in-situ visualization.

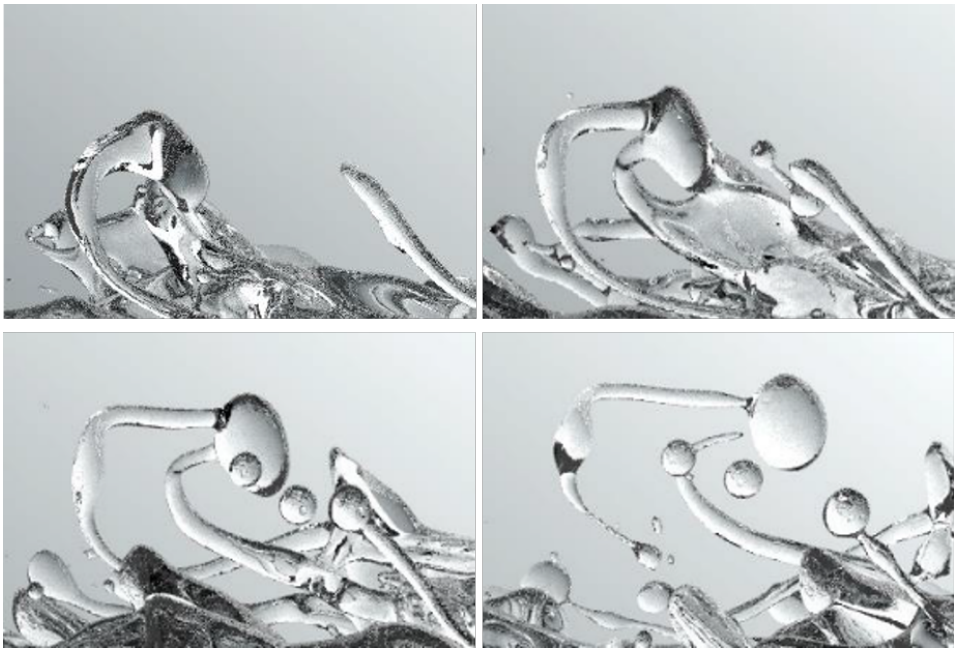


Figure 1: Zoomed view on a droplet evolving in a primary breakup simulation of a multiphase jet using CIAO's multiphase methods [3].

Code and JUQUEEN Performance

We start by focussing on the performance of CIAO. We used a channel setup (Reynolds number 6530) with two periodic directions as target case. In more detail, LES with Lagrangian averaging, 4th order velocity scheme, WENO5 scalar scheme, and 5 (*arts_cf*) or 3 (*arts*) time step subiterations were performed. All considered meshes were uniform and three different case sizes were run (512^3 with 3 passive scalars, 1024^3 with 5 passive scalars, 2560^3 with 10 passive scalars). I/O was avoided (except small monitor files) during the solver iteration and 16 MPI ranks per compute node (each single-threaded) was used.

Scaling of arts_cf

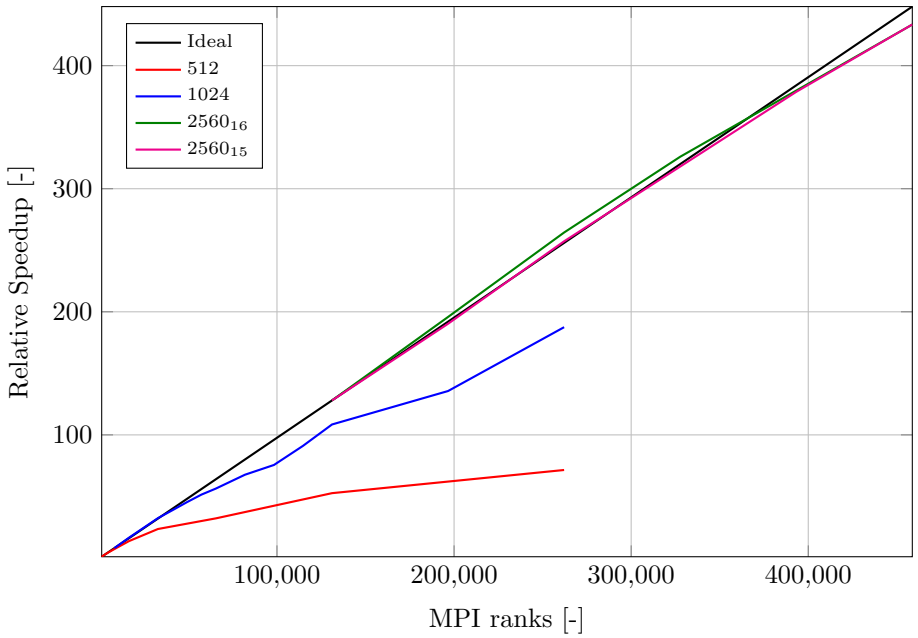
Table 1 shows a summary of all performed scaling runs with respect to *arts_cf* on JUQUEEN. Runs with all three different job sizes were performed and additionally compared to the results of a 2560-case done in August 2015 with a slightly different version of the code (denoted as 2560_{15} in contrast to 2560_{16} for the runs during the scaling workshop). The newer code version has new features with respect to in-situ visualization, Lagrange particles, and a more consistent treatment of passive scalars but should not have significantly different scaling behavior. Within the table, 'C' means that the run crashed due to memory requirements. All new timing measurements are averaged over 15 time steps (whereas 2560_{15} is averaged over 130 time steps) and initialization effects are not considered. The write rate is with respect to writing the simulation state to a single file (resulting file sizes: 512: 44.14 GB, 1024: 421.40 GB, 2560: 9264.64 GB).

Three observations originating from the data in Table 1 should be highlighted:

- The measurements of the cases 2560_{15} and 2560_{16} are almost identical and within typical run-to-run variations (as expected).

Table 1: *arts_cf* strong scaling tests.

bg_size	rpn	MPI ranks	time/step [s] 512	1024	2560_{16}	2560_{15}	write rate [MB/s] 512	1024	2560_{16}
32	16	512	C	C			C	C	
64	16	1024	34.79	C			2610	C	
128	16	2048	16.89	C			1221	C	
256	16	4096	8.87	C			1511	C	
512	16	8192	4.81	45.07			1019	1818	
1024	16	16384	2.55	21.94			1045	2643	
2048	16	32768	1.48	11.14			1311	3753	
2560	16	40960		9.30				3591	
3072	16	49152		7.95				2956	
3584	16	57344		7.00				2255	
4096	16	65536	1.08	6.39	C	C	1050	3623	C
5120	16	81920		5.33				3181	
6144	16	98304		4.77				2973	
7168	16	114688		3.96				2677	
8192	16	131072	0.66	3.32	67.51	67.05	691	3109	12427
12288	16	196608		2.66	44.13	45.10		2750	12437
16384	16	262144	0.49	1.92	32.67	33.35	400	2317	11409
20480	16	327680			26.51				9466
24576	16	393216			22.75	22.66			8036
28672	16	458752			19.94	19.80			4785



- The code scales very well (and up to the full machine) as long as the limit of about 25000 cells per core is not undershot, which is a quite common behavior for compressible flow solvers.

- The code uses MPI-I/O, but the achieved performance does not look good since the data rate decreases for the cases with many cores. Even with Darshan it was not possible to find a reason for the bad MPI-I/O write performance (e.g. the write chunk size, which is often a reason for bad MPI-I/O performance, was found to be good). One reason might be the increasing amount of MPI meta data, which needs to be handled by the root process during the writing, with increasing number of cores. However, even for small cases, it was difficult to evaluate the I/O speed because it was fluctuating a lot. For further analysis, the used JUQUEEN rack should also be considered since racks on JUQUEEN do not have the same number of I/O nodes.

Scaling of arts

For the *arts* scaling study, we considered HYPRE's AMG solver (HYPRE version 2.9.0b for `integer_type=8`) as well as CIAO's BiCGStab method for solving the Poisson equation for the pressure. Although HYPRE's AMG solver is working quite well with CIAO on most clusters, we found some problems using it on JUQUEEN:

- If *arts* was run with the AMG option `extended_interpol=.TRUE.` and about 1024×16 or more MPI-processes (exact lower limit of processes was not determined), the simulation crashed in `HYPRE_BoomerAMGSetup`. Consequently, we used `extended_interpol=.FALSE.` for our simulations as quick workaround. As soon as HYPRE 2.10.1 is available on JUQUEEN, we will check for this bug again and if necessary investigate it in more detail.

- Compared to the BiCGStab method, HYPRE's AMG solver needed more memory, which increased the number of required cores for a fixed problem size due to memory limitations.

- Most of the jobs using 4 or more racks and HYPRE crashed. However, some big jobs with HYPRE ran successfully. For example, it was possible to run an *arts* simulation (2560-case) with HYPRE on the full JUQUEEN, which resulted in similar performance than the corresponding *arts* simulation using BiCGStab. Due to lack of time, the reason for the crashes of some big HYPRE cases was not further investigated.

Because of the problems with HYPRE's AMG solver, we primarily ran the *arts* scaling runs with the BiCGStab solver for the Poisson equation. Results for all three different job sizes are shown in Table 2. The notation is the same as in Table 1. All timing measurements are averaged over 15 time steps and normalized to 10 iterations for the pressure solvers. Initialization effects are not considered. The write rate is with respect to writing the simulation state to a single file (resulting file sizes: 512: 33.39 GB, 1024: 335.45 GB, 2560: 7922.15 GB).

Despite the problems with HYPRE, some *arts* simulations with HYPRE AMG were successfully performed. Their performance (compared to runs with BiCGStab) are additionally summarized in Table 3. Three observations originating from the data in Tables 2 and 3 should be highlighted:

- The BiCGStab solver scales much better than HYPRE's AMG solver for the considered setup.

- The code scales very well on JUQUEEN (and up to the full machine) as long as the limit of about 40000 cells per core is not undershot. This limit is higher than that for the compressible solver due to the elliptic Poisson equation.

- Even though the code needs to write bigger state files for a given case size for *arts_cf*, it requires more runtime memory for *arts*.

Table 2: *arts* strong scaling tests (BiCGStab used).

bg_size	rpn	MPI ranks	time/step [s] 512	1024	2560	write rate [MB/s] 512	1024	2560
32	16	512	C	C		C	C	
64	16	1024	30.9	C		2027	C	
128	16	2048	15.23	C		1120	C	
256	16	4096	7.96	C		1414	C	
512	16	8192	4.24	39.30		1059	1763	
1024	16	16384	2.23	19.15		1149	2230	
2048	16	32768	1.26	9.83		1275	3595	
4096	16	65536	0.90	5.26	C	973	3729	C
8192	16	131072	0.57	2.89	58.22	649	3019	12606
12288	16	196608			37.98			11590
16384	16	262144	0.42	1.64	28.29	384	2235	11277
20480	16	327680			22.75			8952
24576	16	393216			19.63			7750
28672	16	458752			17.11			6637

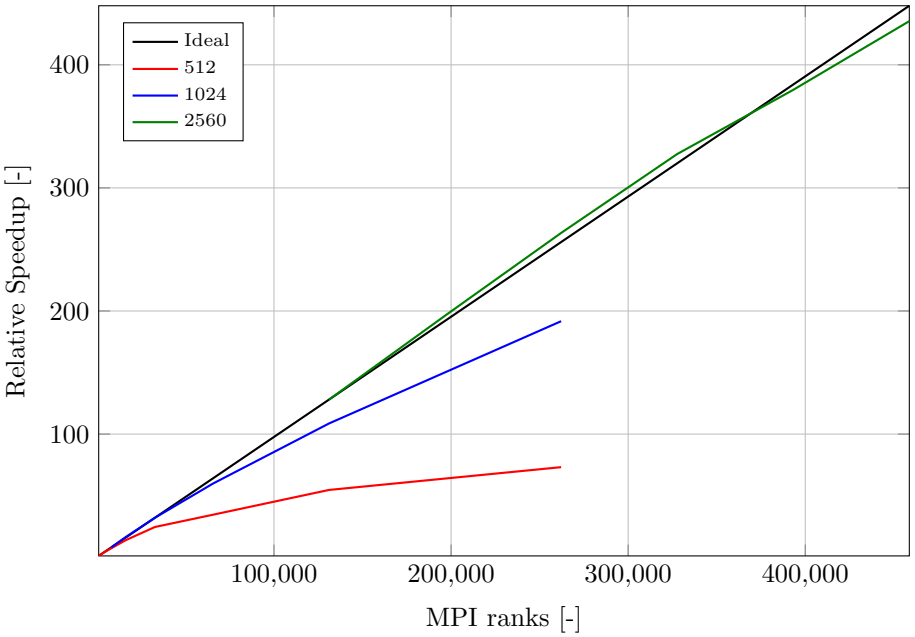


Table 3: Overview of performance of HYPRE's AMG solver compared to the BiCGStab solver. Values are the time for 15 time steps with HYPRE's AMG divided by the corresponding time using the BiCGStab solver. 'CH' indicates that the run with HYPRE crashed while it was successful with BiCGStab. 'C' means that both runs crashed.

Case	MPI ranks 512	1024	2048	4096	8192	16384	32768	65536	131072	262144
512	C	0.97	1.00	1.01	1.03	1.17	1.55	2.00	CH	CH
1024	C	C	C	C	CH	0.99	1.04	1.08	CH	CH

JUQUEEN Performance

In order to understand the advantages and shortcomings of JUQUEEN compared to other clusters, we ran the 512-channel case (*arts* with AMG and *arts_cf*) on four different clusters: JSC-JUQUEEN, RWTH-BULLW, ITV-NEHAL, and ITV-OXYFLAME. Among those, JSC-JUQUEEN is by far the biggest machine, but has the smallest amount of memory per core. The configuration of the different clusters is summarized in Table 4. The performance of each cluster should be determined by the used CPUs, memory, and network. Since *arts* needs to solve an elliptic Poisson equation for the pressure in each subiteration, it is expected to benefit more from a good network than *arts_cf*.

Table 4: Overview of cluster configurations.

	JSC-JUQUEEN	RWTH-BULLW	ITV-NEHAL	ITV-OXYFLAME
CPUs per node	IBM PowerPC A2	2x Intel Xeon X5675	2x Intel Xeon X5670	2x Intel Xeon E5-2660v2
Clockspeed	1.6 GHz	3.06 GHz	2.93 GHz	2.20 GHz
Cores per node	16	12	12	20
SDRAM type	DDR3-1333	DDR3-1333	DDR3-1333	DDR3-1866
Memory per node	16 GB	24 GB	48 GB	256 GB
Network	5D Torus - 40 GBps	Infiniband QDR	Infiniband QDR	Infiniband FDR-10
CCT(*arts*) [core-h]	17.24		5.28	3.55
CCT(*arts_cf*) [core-h]	9.90	2.32	2.05	1.46

Table 5: Average time/step [s] on different clusters.

JSC-JUQUEEN			RWTH-BULLW		ITV-NEHAL			ITV-OXYFLAME		
ranks	*arts_cf*	*arts*	ranks	*arts_cf*	ranks	*arts_cf*	*arts*	ranks	*arts_cf*	*arts*
512	C		672	C	144	C	C	40	C	C
1024	34.79	60.62	720	11.62	192	38.48	C	60	87.35	C
2048	16.89	31.17	768	11.11	240	31.41	79.23	80	66.12	159.85
4096	8.87	18.25	816	10.81	288	26.83	64.37	100	53.33	119.04
8192	4.81	11.49	864	9.92	336	23.14	56.64	120	45.15	100.70
16384	2.55	8.62	912	10.17						
32768	1.48	9.93	960	9.14						
65536	1.08	11.33								
131072	0.66	C								
262144	0.49	C								

The results of the runs on the different clusters are shown in Table 5 (notation same as in Table 1). Due to the smallest memory per core, JUQUEEN's runs are typically much bigger in terms of used total cores than those of the other clusters. However, because of the good scaling properties of CIAO, comparisons are still fair. For each cluster and both solvers, we computed the computing cost of each time step CCT defined as number of cores used times average wall time per time step. It measures the needed amount of core-h of each system to advance the test case by one time step. The computing cost per time step is expected to be constant for runs with different number of cores on the same cluster as long as the code scales perfectly with respect to the case and the used number of cores. Thus, we computed for each cluster and each solver one computing cost per time step value, which is also included in Table 5. It can be seen that using *arts_cf* on JSC-JUQUEEN about 6.78 times more core-h are needed for the same simulation than on ITV-OXYFLAME due to the slower clockspeed of the CPUs. Since the network (which is fastest on JUQUEEN) is more important for *arts*,

this factor decreases for *arts* to about 4.86.

MPI-I/O and `romio.hints`

Since we found MPI-I/O to be critical for the performance of the code, we tried to study the used ROMIO settings. In more detail, exact romio hints were given to simulations by exporting the corresponding environmental variable (`ROMIO_HINTS`) in the job submission script. During execution we checked the actually used ROMIO hints with respect to the written data file with the code shown in Listing 1.

Listing 1: Code for printing `romio.hints`.

```
! Declare your variables
logical :: hintflag, romiohints
integer :: hintsused, keys, nkeys, info
character(len=255) :: key, value

!
! Use MPI_FILE_OPEN with file handle fh to open the file for I/O
!

! Print the romio.hints
if ((irank == iroot) .AND. romiohints) then
    call MPI_FILE_GET_INFO(fh,hintsused,info)
    call MPI_INFO_GET_NKEYS(hintsused, nkeys, info)
    do keys=0,nkeys-1
        call MPI_INFO_GET_NTHKEY(hintsused, keys, key, info)
        call MPI_INFO_GET(hintsused, key, 255, value, hintflag, info)
        if(hintflag) write(*,*) 'ROMIOHINTS: ', trim(key), " = ", trim(value)
    end do
end if
```

As test case we used the compressible 1024-channel setup with 10 passive scalars. The GPFS filesystem type was specified with `BGLOCKLESSMPIO_F_TYPE` always set to 0x47504653. In `romio.hints`, we set the following variables to different values in different simulations:

- cb_buffer_size (default: 33554432)
- romio_cb_read ('enable')
- romio_cb_write ('enable')
- cb_nodes (4160)
- romio_no_indep_rw ('FALSE')
- romio_cb_pfr ('disable')
- romio_cb_fr_types ('aar')
- romio_cb_fr_alignment (1)
- romio_cb_ds_threshold (0)
- romio_cb_alltoall ('automatic')
- ind_rd_buffer_size (4194304)
- ind_wr_buffer_size (???)
- romio_ds_read ('automatic')
- romio_ds_write ('disable')

During execution we were just able to change the value for `romio_ds_read` from 'automatic' to 'enable' (without significant impact on the write rate). All other variables remained always the same in the simulation regardless of what we set in the `romio.hints` file

Each set of parameters was run twice (once during the workshop and once after the workshop). While the write rates of the first runs (run during the workshop) differed by only 3.5% of the average write rate (3504 MB/s), those of the second runs (run after the workshop in normal JUQUEEN operation) differed by about 27.2% of the average write rate (3430 MB/s).

In-Situ Visualization

As part of the JARA-HPC project JHPC18 [4, 5], methods for in-situ visualization of large simulations are developed in order to allow easy and fast control of large simulations and reduce the amount of data which needs to be stored. More precisely, the coupling layer JUSITU (JUSITU's work flow is shown in Figure 2 [1]) has been implemented and coupled to CIAO and VisIt [2]. One goal for the scaling workshop was to show the usability of JUSITU on the full JUQUEEN. Since the target applications of the JARA-HPC project are multiphase flows, we chose a compressible, turbulent channel (Reynolds number 13,760) containing small droplets (described by an Eulerian indicator function) as test case and ran it successfully on 1, 4, 16, 20, and 28 racks. Figure 3 shows a screenshot of the full JUQUEEN run. It visualizes the simulation state after running for 9 units of simulation time. Beside the *VisIt* overview window (on the left), *Window 1* showing a histogram of the pressure, *Window 2* visualizing the turbulent kinetic energy within the channel, the *Compute engines* window giving information about the simulation on JUQUEEN, and the *Simulation* window allowing to give instructions to the simulation are visible.

Miscellaneous

Since CIAO is not hybrid-parallelized yet, we tested auto-parallelization and ran CIAO with 2 threads per process. More precisely, we compiled CIAO with the IBM XL compiler `-qsmp=auto` flag. However, we found that just using the compiler auto-parallelization the performance dropped by about 40% compared to a run with 1 tpp.

Conclusions

The workshop helped us to better understand the performance of the CIAO code. Additionally, it enabled us to successfully run JUSITU on the full machine. Since running JUSITU interactively on the full JUQUEEN was not trivial, several tries were needed until all bugs and problems were solved (e.g. with respect to memory management). This would not have been possible without the simple access to the full machine during the scaling workshop. Because JUSITU can be simply used also with other codes, we expect that many JUQUEEN users will benefit from our experience during the workshop in the future.

Acknowledgments

The authors gratefully acknowledge the computing time (JHPC18) granted by the JARA-HPC Vergabegremium and provided on the JARA-HPC Partition part of the supercomputer JUQUEEN [6] at Forschungszentrum Jülich. Furthermore, the authors gratefully acknowledge funding by the Cluster of Excellence "Tailor-Made Fuels from Biomass" and Honda R&D.

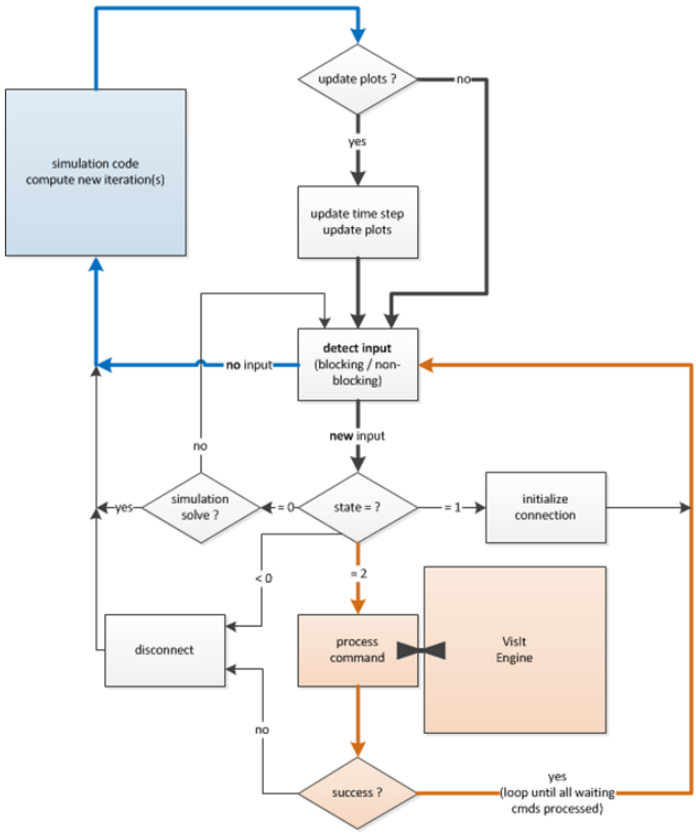


Figure 2: Flow chart of JUSITU workflow.

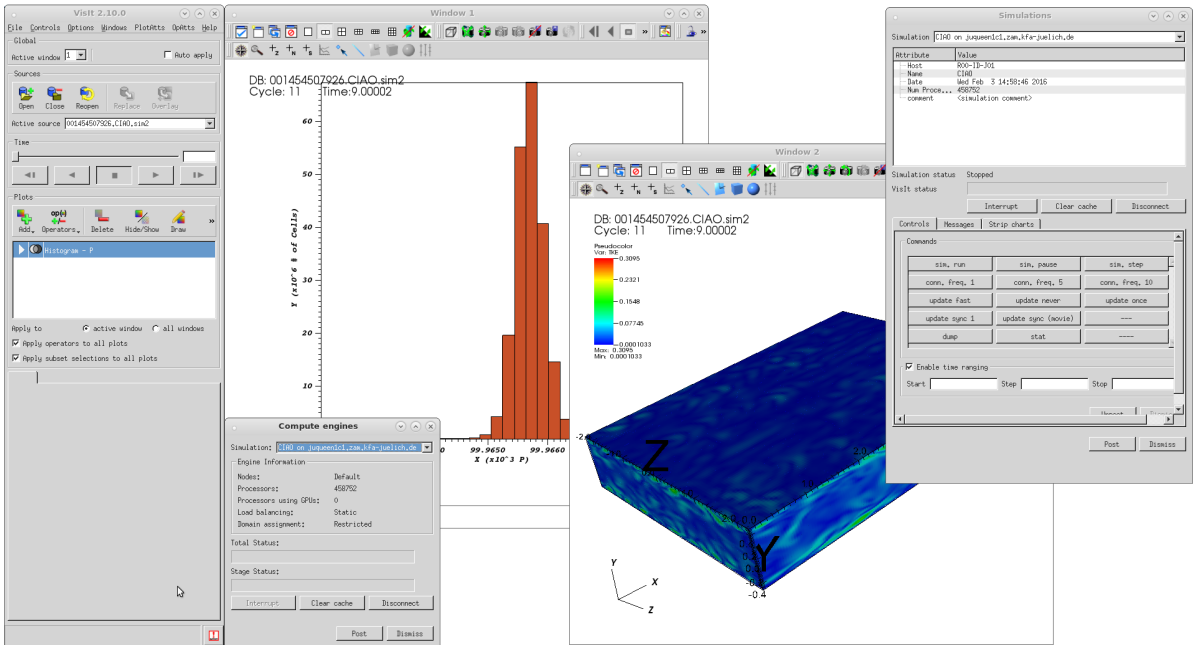


Figure 3: Screenshot of the visualization client showing successful CIAO/JUSITU/VisIt run on full JUQUEEN (458,752 MPI processes).

References

[1] https://trac.version.fz-juelich.de/vis/wiki/VisIt

[2] https://wci.llnl.gov/simulation/computer-codes/visit

[3] M. Bode, J.H. Göbbert, H. Pitsch; *Novel multiphase simulations investigating cavitation by use of in-situ visualization and Euler/Lagrange coupling*; PRACEdays15, Dublin (2015)

[4] M. Bode, J.H. Göbbert, H. Pitsch; *Detailed Investigation of Liquid Sheet Breakup Using Direct Numerical Simulation and In-situ Visualization*; JARA-HPC Proposal (2014,2015)

[5] M. Bode, J.H. Göbbert, H. Pitsch; *High-fidelity multiphase simulations and in-situ visualization using CIAO*; NIC Symposium, Jülich (2016)

[6] Jülich Supercomputing Centre; *JUQUEEN: IBM Blue Gene/Q Supercomputer System at the Jülich Supercomputing Centre*; Journal of large-scale research facilities **1** (2015) A1; http://dx.doi.org/10.17815/jlsrf-1-18

Performance of Code_Saturne at scale on JUQUEEN

Charles Moulinec[1], Yvan Fournier[2], Vendel Szeremi[1], and David R. Emerson[1]
[1]STFC Daresbury Laboratory, UK
[2]EDF R&D, France

Description of the Code

Code_Saturne [1–3] is an open-source CFD software package based on the finite volume method to simulate the Navier-Stokes equations. It can handle any type of mesh built with any cell/-grid structure. Incompressible and compressible flows can be simulated, with or without heat transfer, and a range of turbulence models are also available. The velocity-pressure coupling is handled using a projection-like method. The default algorithm to compute the velocity is the Jacobi algorithm and the pressure is solved with the help of an algebraic multigrid (AMG) algorithm. Parallelism is handled by distributing the domain over MPI processes, with an optional second level of shared memory parallelism based on the OpenMP model. Several partitioning tools are available, i.e. geometric ones (Space-Filling Curve with Morton and Hilbert approach) and graph-based ones (METIS, ParMETIS, SCOTCH and PT-SCOTCH).

Code_Saturne can be used as a standalone package, but extra libraries may also be plugged in, as to read some of the supported mesh formats (CGNS, MED, CCM, for instance), to get access to graph-partitioners (METIS, ParMETIS, SCOTCH, PT-SCOTCH) or to additional sets of linear solvers (PETSc, for instance). In the first part of this work, *Code_Saturne* is used as a standalone package, and then uses PETSc.

The code (350,000 lines) is written in Fortran (∼37%), C (∼50%) and Python (∼13%). In this work, Python is only used on the frontend, where *Code_Saturne* 'cases' are prepared, i.e. the executable is created, accounting from the user subroutines, and a symbolic link is added pointing on the mesh in the native format.

MPI is used for communication between subdomains and OpenMP pragmas have been added to the most time-consuming parts of the code.

MPI-IO is used for output of postprocessing files, which by default uses the EnSight Gold format, also readable by ParaView, for dumping potential checkpointing files and meshes (*mesh_output* file) if requested by the user, and also for reading the *mesh_input* file and potential restart files.

The recent developments of the code to join [2] or to globally refine meshes in parallel [4] are exploited to obtain large domain meshes (several billion cells) directly on the machine where the calculation takes place.

Figure 1 shows a snapshot of the velocity magnitude computed in the FDA blood pump test case [5].

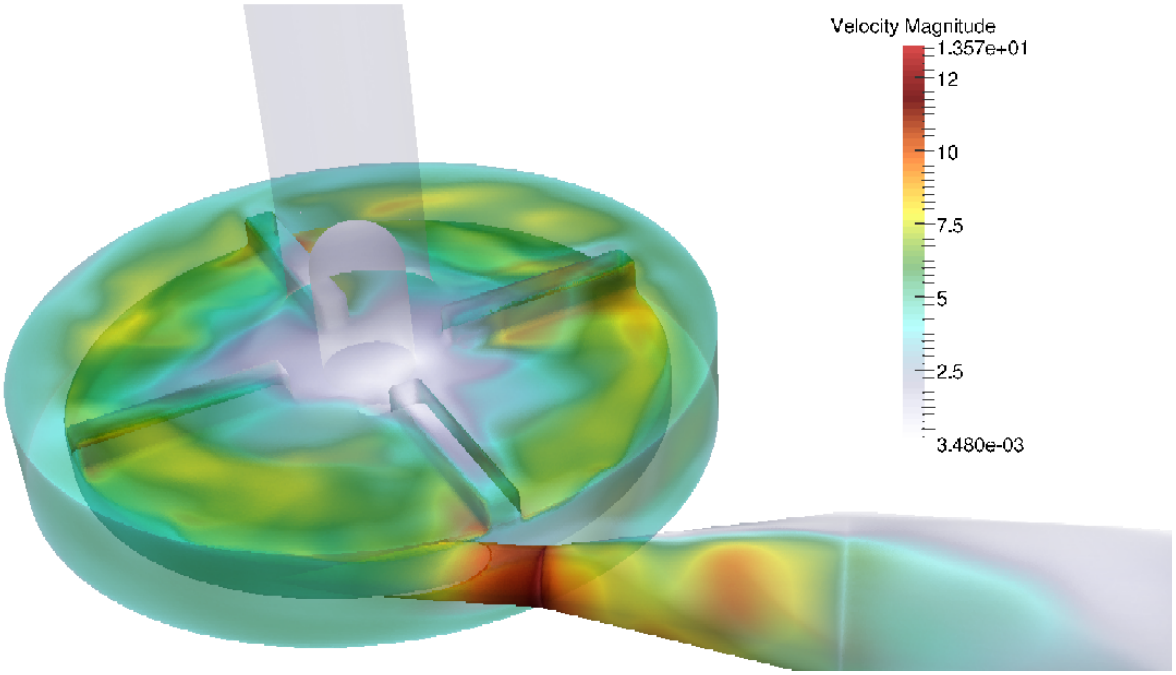


Figure 1: Snapshot of the velocity magnitude in a blood pump.

Objectives - Test case

The objectives of the Extreme Scaling Workshop were to:

1. port *Code_Saturne* version V4+ on JUQUEEN,
2. test at scale the performance of the hybrid implementation MPI/OpenMP of the code for a mesh based on tetrahedral cells, in 2 configurations, to compute the pressure Poisson equation, using either the algebraic multigrid (AMG) algorithm as a solver, preconditioned by the diagonal of the matrix (*Diagonal+AMG*) or the AMG algorithm as a preconditioner with a conjugate gradient (CG) solver (*AMG+CG*),
3. test the performance of the AMG solver of PETSc.

The test case is the classical flow in a lid-driven cavity. The mesh has been generated by Mesh Multiplication (also called Global Mesh Refinement) to get a 7 billion tetrahedral cell mesh. This was done during the workshop, but with an earlier version of the code.

File input/output were not to be tested, but their performance to read the 618 GiB mesh is listed in Table 1. Reading the mesh takes less time than performing a time step for all the simulations carried out here. We decided not to write any large (postprocessing and checkpointing) files to disk during the workshop to shorten the time for the simulations to complete.

Results

Code_Saturne version 4.2.0 was successfully ported on JUQUEEN, and linked to the PETSc library `/bgsys/local/petsc/3.6.0/juqueen-downloads-O3strict_int8`

Table 1: IO performance reading the 618 GiB *mesh_input* file.

Racks	rpn	MPI ranks	tpp	Threads	IO reading time [s]
4	16	65,536	1	65,536	49.68
4	16	65,536	2	131,072	46.00
4	16	65,536	4	262,144	44.74
8	16	131,072	1	131,072	30.91
8	16	131,072	2	262,144	26.99
8	16	131,072	4	524,288	36.03
16	16	262,144	1	262,144	32.29
16	16	262,144	2	524,288	29.31
16	16	262,144	4	1,048,576	28.26
28	16	458,752	1	458,752	44.51
28	16	458,752	2	917,504	41.49
28	16	458,752	4	1,835,008	41.34

The first tests using *Code_Saturne*'s native storage format were not conclusive, i.e., the number of sub-iterations for the linear solvers to converge was increasing with the number of OpenMP threads used per node. Switching to the Modified Compressed Sparse Row (MSR) format fixed the issue and the number of sub-iterations were kept constant whatever the number of OpenMP threads used. Changing for the MSR format had minimal impact on the memory consumption.

The optimal number of ranks per node was 16, as at least 1 GiB memory was required per MPI task, due to the memory required to partition the mesh using the Hilbert Space Filling Curve algorithm.

Table 2 shows the timing observed when using AMG as a solver (*Diagonal+AMG*) and as a preconditioner (*AMG+CG*), respectively. The first 5 time steps of the simulation were carried out, as this is enough to assess the behaviour of the code. Note that a simulation might last thousands or hundreds of thousands time steps, depending on the physics used. *Code_Saturne* was tested on 4, 8, 16 and 28 racks, and for each of them using 1, 2 or 4 OpenMP threads and 16 ranks per node. All the simulations show that a speed-up is observed when going from 1 to 2 and then 4 OpenMP threads, making good use of all the hardware threads available per compute node processor.

Figure 2 shows the time per time step, using the AMG as a solver (*Diagonal+AMG*) and as a preconditioner (*AMG+CG*), and 4 OpenMP threads and 16 ranks per node. Good performance is observed.

The tests carried out using the PETSc library were inconclusive, because *Out of Memory* issues were observed when using more than 2 ranks per node, whatever the size of the mesh (three meshes were tested, using 111 M cells, 889 M cells and 7 B cells respectively). This behaviour has to be further investigated.

Table 2: Strong scaling tests of *Diagonal+AMG* and *AMG+CG*.

Racks	rpn	MPI ranks	tpp	Threads	Time per time step [s]	
					Diagonal+AMG	*AMG+CG*
4	16	65,536	1	65,536	394.05	363.18
4	16	65,536	2	131,072	255.75	230.93
4	16	65,536	4	262,144	255.21	190.86
8	16	131,072	1	131,072	251.05	214.24
8	16	131,072	2	262,144	173.39	143.75
8	16	131,072	4	524,288	148.36	117.30
16	16	262,144	1	262,144	131.55	110.80
16	16	262,144	2	524,288	95.76	76.17
16	16	262,144	4	1,048,576	81.01	62.63
28	16	458,752	1	458,752	81.60	67.44
28	16	458,752	2	917,504	61.84	48.07
28	16	458,752	4	1,835,008	55.66	41.23

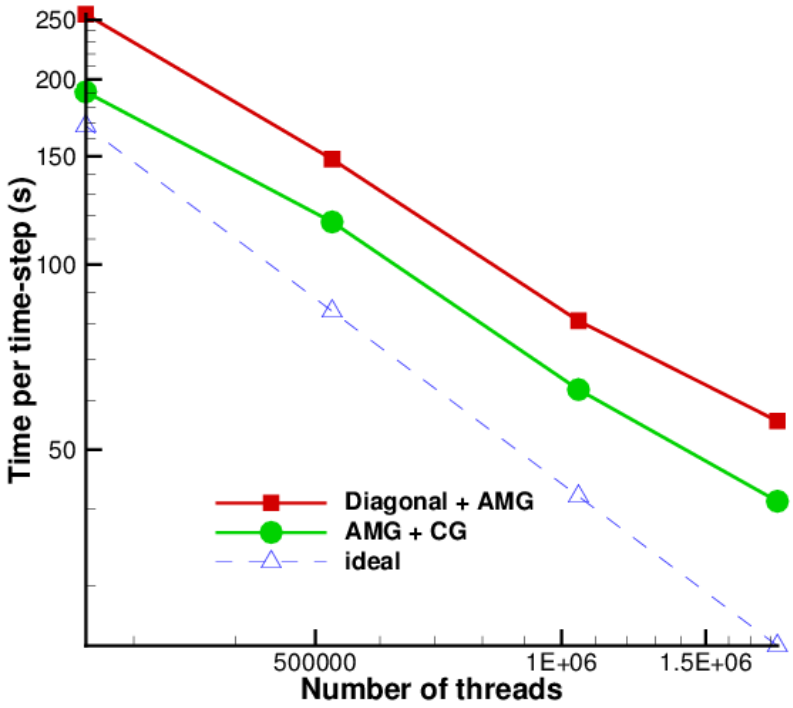


Figure 2: Time per time step as a function of the number of threads (4 OpenMP threads for each of 16 MPI tasks per compute node): *Diagonal+AMG* vs *AMG+CG*.

Conclusions

Simulations using tetrahedral meshes were carried out at scale, with the latest version of *Code_Saturne* (V4.2.0), in order to assess the performance of the hybrid MPI/OpenMP implementation. Using the MSR format clearly helps, as a speed-up is observed for a fixed number of ranks per node (16) when increasing the number of threads from 1 to 4. Attending the workshop was extremely beneficial, as similar tests carried out with an earlier version of the code (V3.0.0) three years ago, on MIRA (DOE - Argonne) did not exhibit any speed-up.

References

[1] F. Archambeau, N. Méchitoua, M. Sakiz; *Code_Saturne: a finite volume code for the computation of turbulent incompressible flows - industrial applications*. International Journal on Finite Volumes **1** (2004) 1–62.

[2] Y. Fournier, J. Bonelle, C. Moulinec, Z. Shang, A.G. Sunderland, J.C. Uribe; *Optimizing Code_Saturne computations on Petascale systems*. Computers & Fluids **45** (2011) 103–108.

[3] `http://www.code-saturne.org`

[4] A, Ronovsky, P. Kabelikova, V. Vondrak, C. Moulinec; *Parallel mesh multiplication and its implementation in Code_Saturne*. B.H.V. Topping and P. Iványi (editors), Third PARENG, Civil-Comp Press, Stirlingshire, UK (2013).

[5] V. Marinova, I. Kerroumi, A. Lintermann, J.-H. Göbbert, C. Moulinec, S. Rible, Y. Fournier, M. Behbahani; *Numerical analysis of the FDA centrifugal blood pump*. NIC Symposium 2016 (2016).

Extreme Scaling of IciPlayer with Components: IciMesh and IciSolve

Hugues Digonnet
Institut de Calcul Intensif, École Centrale de Nantes, 1 rue de la Noë, 44300 Nantes, France

Description of the Code

The IciPlayer code is dedicated to numerical simulations based on an implicit finite element formulation including anisotropic mesh adaptation using an error estimator. The code is written in C++ and parallelized using pure MPI and does not yet use multi-threading functionalities. It is based on component assembly to generate a specific application. The main components tested during this Extreme Scaling Workshop were:

- IciMesh: anisotropic mesh adaptation using an error estimator and topological improvement. The parallel version of the mesher is obtained executing the sequential one independently on each subdomain under the constraint of keeping the interfaces between subdomains unchanged. Then, a mesh repartitioner is used to migrate interfaces to inner parts to be remeshed during the next step of an iterative procedure.

- IciSolve: a multigrid solver. For that, we have used the multigrid framework provided by the PETSc library. For each level, we need to build the system to be solved and also the restriction/interpolation operator between two consecutive levels. Thanks to the mesh adaptation strategy, these operators are mainly local with only few external non-zero values.

- IciIO: an input/output module. The default configuration being to write and read one file per MPI process and a new MPI-IO version has also been implemented to create only one single shared file. An usual file size is between 4 and 10 MB per MPI process.

Figure 1 presents an application of the code for wind turbine simulations (top) and also the partition of a 33.4 billions nodes mesh (bottom), done over 65 536 cores. More illustrations and details may be found on the website [1] and in [2].

Results

The goal when participating at the Extreme Scaling Workshop was to evaluate the scalability of our code on a full Tier0 supercomputer (like JUQUEEN) scale. The benchmark used to test it consists in executing only one step of the two main CPU consuming parts (more than 90% for both) of a usual simulation run:

- mesh adaptation (IciMesh)

- linear system resolution (IciSolve)

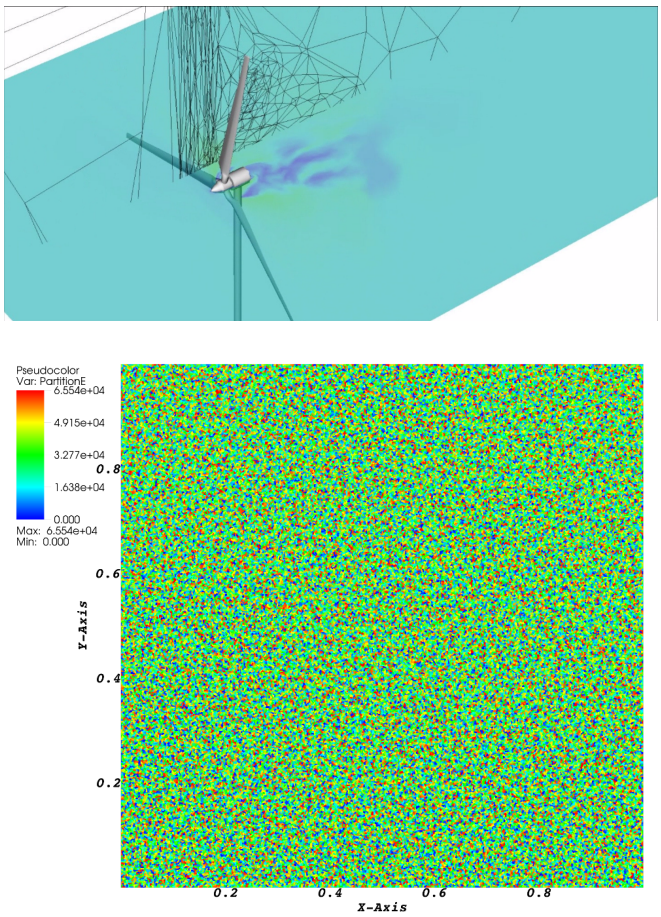


Figure 1: Illustrations of IciPlayer: 3d computation of the flow around a moving wind turbine (top); partition of a 33 billion node mesh of a 2d unit square into 65 536 subdomains (bottom).

First IciMesh is used to generate several meshes that will be given as an input for the multigrid solver, IciSolve. Due to limited time during this 3-day workshop (30 minutes during the day and less than 2 hours per participant during the night) only two dimensional benchmarks have been considered. The results presented below show performance achieved for both components, IciMesh and IciSolve, but also point out some implementation/configuration problems found in the MPI-IO input/output functions.

IciMesh

The first benchmark done during the workshop consisted mainly in the generation of very large meshes. For that, the refinement of an initial mesh as been done by dividing the initial homogeneous mesh size by a factor of 4, leading to a mesh sixteen times larger. Weak scaling performance was analysed and thus the time spent to generate the final mesh remained constant. Runs have been executed using two series: the first one started with a 6 250 nodes/process mesh to generate a 100 000 nodes/process one, using 1, 4, 8, 16 racks; the second one started with a 7 200 nodes/process mesh to generate a 115 000 nodes/process one, over 1.75 and 28 racks. This second benchmark corresponds to the 2 and 32 racks configuration of the first one, but using 'only' 1.75 and 28 racks (for the full JUQUEEN system). Table 1

represents the initial and final mesh size, in number of nodes, for all the tested configurations.

Table 1: IciMesh: Size of the initial and final meshes in nodes and GB. (* for the 28 racks run, the mesh was not written to disk)

MPI ranks	Initial (nodes)	Final (nodes)	Initial (GB)	Final (GB)
16384	104 286 789	1 665 319 673	6.5	100
65536	417 097 385	6 661 140 625	25	400
131072	817 567 546	13 056 197 022	50	800
262144	1 668 284 934	26 644 028 546	100	1 600
28672	204 389 553	3 263 946 831	13	200
458752	3 269 500 345	52 219 945 522	200	*

Tables 2 and 3 present the time needed to execute the mesh adaptation procedure, summarized also in Figure 2. These results firstly confirm the linear complexity of the mesh adaptation algorithm (CPU time increased by 15% between series one and two), but also the really good parallel efficiency of the parallelisation strategy, which keeps the remeshing time almost constant, whatever the number of MPI processes used. The efficiency is still 0.91 using 16 racks, and 0.88 using all 28 racks of JUQUEEN. Figure 3 presents the almost perfect speed-up, in terms of number of nodes created per second during the adaptation procedure, from 1 to 28 racks of JUQUEEN.

Table 2: IciMesh: weak scaling tests (s1: 100 000 nodes/MPI process). Time to adapt the mesh and write it to disk.

bg_size	rpn	MPI ranks	tpp	Threads	Compute (s)	Output (s)
1024	16	16384	1	16384	202.60	60.38
4096	16	65536	1	65536	208.72	56.14
8192	16	131072	1	131072	215.07	86.31
16384	16	262144	1	262144	223.73	115.75

Table 3: IciMesh: weak scaling tests (s2: 115 000 nodes/MPI process). Time to adapt the mesh and write it to disk. (* for the 28 racks run, the mesh was not written to disk)

bg_size	rpn	MPI ranks	tpp	Threads	Compute (s)	Output (s)
1792	16	28672	1	28672	230.58	60.74
28672	16	458752	1	458752	263.33	*

IciSolve

Here, the scalability of the multigrid solver is evaluated. We have chosen the resolution of the Stokes equation using a finite element method and a mixed formulation for the velocity/-pressure unknowns, given by the element P1+/P1. Although the results presented in Table 4

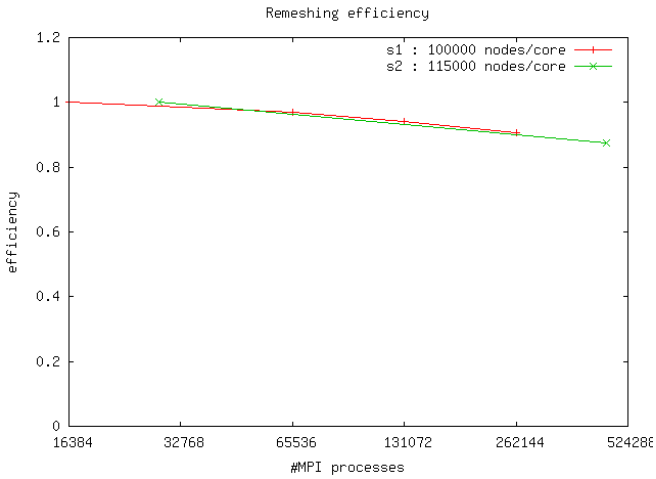


Figure 2: Parallel efficiency of IciMesh for the two series, s1 and s2, using 1 to 28 JUQUEEN racks.

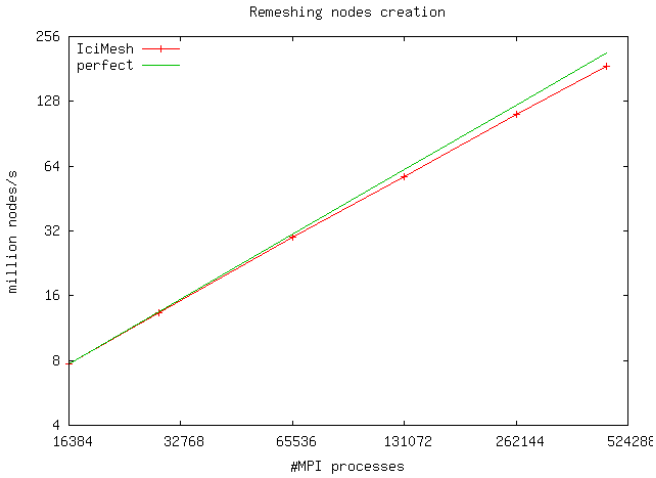


Figure 3: Speed-up in terms of million of nodes created during the mesh adaptation procedure, compared also to the perfect scaling one.

are very encouraging, we have not been able to test the scalability above 4 racks due to the MPI-IO problem (see Table 5 of the IciIO section). The parallel efficiency for the resolution step is perfect (1.03), but is not as good (0.81) for the assembly of linear systems, as seen in Table 4 (assembling times). This may need some further investigation to understand if this is due to PETSc [3] options or directly to our code. Looking at the global time to compute the solution, we observe that it goes from 126.38 seconds on one rack to 133.42 seconds using 4 racks, which gives an efficiency of 0.95.

IciIO

This test was not planned at the beginning of the workshop, but it appeared when the MPI-IO input/output implementation became the main obstacle using the 28 racks of JUQUEEN. Table 5 presents the time spent in reading all the meshes of the different levels used to build

Table 4: IciSolve: weak scaling tests. *Comment:* time excluding IO that will be discussed in the IciIO section. (* the run reach the one hour allocated time limit before ending reading mesh files)

bg_size	rpn	MPI ranks	tpp	Threads	Time (s)
assembling the finest level					
1024	16	16384	1	16384	22.66
4096	16	65536	1	65536	28.13
16384	16	262144	1	262144	*
assembling all levels (including the finest one)					
1024	16	16384	1	16384	25.04
4096	16	65536	1	65536	34.49
16384	16	262144	1	262144	*
solving linear system					
1024	16	16384	1	16384	99.90
4096	16	65536	1	65536	97.39
16384	16	262144	1	262144	*
neglected					
1024	16	16384	1	16384	1.44
4096	16	65536	1	65536	1.54
16384	16	262144	1	262144	*
total time (without IO)					
1024	16	16384	1	16384	126.38
4096	16	65536	1	65536	133.42
16384	16	262144	1	262144	*

the large linear system. This table, when compared to Table 4, shows that almost all the time spent in the IciSolve test is spent in the MPI-IO reading operation. It also tells us that this implementation does not scale and we have reached the 60-minute time limit of the job while using 16 racks. Unfortunately, we have not managed, during the workshop, to use Darshan (as no output report was generated) to finely analyse what happened during IO operations. A post workshop analysis seems to point out the fact that we use the individual `MPI_File_read` function rather than collective `MPI_File_read_all` when the number of sub-domains in the mesh partition is smaller than the number of MPI processes used (still under investigation).

Even with this poor performance obtained for IO, their proportion, compared to the benchmarks' times, is not representative of a typical simulation run, where we save files only every 10 to 100 increments and represent less than 5% of the total simulation time.

Table 5: IciIO: weak scaling tests (s1: 100 000 nodes/MPI process). Mesh file sizes and reading times in the IciSolve benchmark test.

bg_size	rpn	MPI ranks	tpp	Threads	Time (s)	size (GB)
1024	16	16384	1	16384	645	110
4096	16	65536	1	65536	950	440
16384	16	262144	1	262144	>3 600	1 760

Conclusions

We have been able to use the 28 racks of JUQUEEN with IciMesh with a really good efficiency (0.88), leading to the generation of an adapted mesh containing 52 billion nodes and 104 billion elements. Performance obtained using the multigrid solver of IciSolve is very encouraging but has only been tested up to 4 racks due to the main difficulty encountered during this workshop with the MPI-IO input/output. The implementation/configuration of MPI-IO will be analysed and rewritten shortly to enable doing file input/output while using more than 100 000 MPI processes.

Some additional benchmarks may be done on BlueGene/Q using more (2 or 4) MPI processes per core to really exploit the PowerPC architecture. As soon as the MPI-IO problem can be solved, benchmarks of IciSolve may be done above 4 racks. Future investigations will also be performed to understand the loss of efficiency during the assembly step, which may become an obstacle to the scalability of IciSolve, even if it usually represents around 25% of the implicit finite element resolution.

Acknowledgments

We acknowledge the organisers of the Extreme Scaling Workshop for the access to JUQUEEN supercomputer and their expertise, as well as Fabien Delalondre who asked me to participate; but also GENCI and PRACE for enabling me, during the last 7 years, to improve the parallel efficiency of the algorithms developed; without omitting Luisa Silva, Thierry Coupez, Patrice Laure and all other team members that actively contribute to the code.

References

[1] `http://ici.ec-nantes.fr/`

[2] H. Digonnet, L. Silva, T. Coupez; *Using Full Tier0 Supercomputers for Finite Element Computations with Adaptive Meshing*; in P. Iványi and B.H.V. Topping (editors), Proceedings of the Fourth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering, Civil-Comp Press, Stirlingshire, UK, Paper 13, 2015; [doi:10.4203/ccp.107.13]

[3] S. Balay *et al*; *PETSc user manual, Revision 3.1*; Argonne National Laboratory, ANL-95/11, 2010;

iFETI: Scalable Nonlinear FETI-DP/Multigrid Methods

Axel Klawonn[1], Martin Lanser[1], and Oliver Rheinbach[2]
[1]Mathematisches Institut, Universität zu Köln,
Weyertal 86-90, 50931 Köln, Germany.
E-mail:{axel.klawonn,martin.lanser}@uni-koeln.de
[2]Institut für Numerische Mathematik und Optimierung,
Fakultät für Mathematik und Informatik,
Technische Universität Bergakademie Freiberg,
Akademiestr. 6, 09596 Freiberg, Germany.
E-mail: oliver.rheinbach@math.tu-freiberg.de

Description of the Code

Description of the Method

We are concerned with the development of highly scalable implicit solvers for finite element problems in nonlinear hyperelasticity and plasticity problems. These solvers [1] constitute the computational kernel for our finite element simulation environment. Previously [2], we have scaled a highly concurrent multiscale method, combined with domain decomposition solvers, to the complete JUQUEEN supercomputer [3]. In this paper, we focus on the implicit solver applied to standard finite element discretization.

Here, we consider a variant of a recent nonlinear FETI-DP (inexact nonlinear FETI-DP; iNL-FETI-DP) domain decomposition method; see [4, 5]. As in the linear inexact FETI-DP method [6–8], in the iNL-FETI-DP method, we combine an algebraic multigrid method with a FETI-DP type domain decomposition. In this approach, we now completely remove sparse direct solvers from the algorithm. This will typically allow us to use larger subdomains.

Thus, we strive to combine the robustness of FETI-DP methods for structural mechanics problems, with the increased concurrency of nonlinear domain decomposition methods and the efficiency of multigrid methods.

Our approach is based on the solution of the nonlinear FETI-DP saddle point system

$$\begin{array}{ll} \widetilde{K}(\tilde{u}) + B^T\lambda - \tilde{f} & = 0 \\ B\tilde{u} & = 0 \end{array} \tag{1}$$

using Newton's method, which leads to linearized systems of the form

$$\begin{bmatrix} D\widetilde{K}(\tilde{u}) & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \delta\tilde{u} \\ \delta\lambda \end{bmatrix} = \begin{bmatrix} \widetilde{K}(\tilde{u}) + B^T\lambda - \tilde{f} \\ B\tilde{u} \end{bmatrix}; \tag{2}$$

see also [9] for a detailed description of nonlinear FETI-DP methods. The bigger part of $D\widetilde{K}(\tilde{u})$ is a block diagonal matrix and each diagonal block $(DK_{BB}(\tilde{u})^{(i)})$ is associated to one FETI-DP subdomain. The remaining part is obtained from a global coupling in few variables and constitutes the FETI-DP coarse problem. The jump operator B ensures continuity on

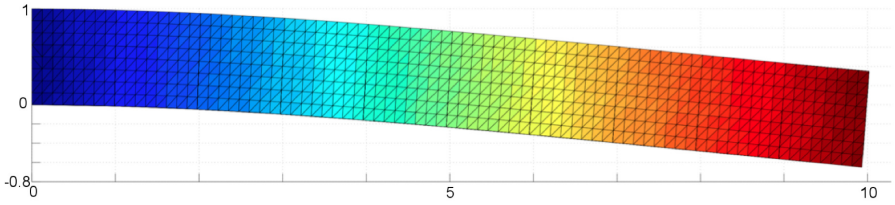


Figure 1: **Linear 2D beam**; elasticity problem on a rectangular domain; fixed on the left; volume force applied in vertical direction.

the interface between the subdomains and contains only one 1 and one -1 per row. Thus, a multiplication of B with a vector only causes nearest neighbor communication. The saddle point system (2) is solved iteratively with GMRES using the block-triangular preconditioner

$$\mathcal{B}_L := \begin{bmatrix} \widehat{K}^{-1} & 0 \\ M^{-1}B\widehat{K}^{-1} & M^{-1} \end{bmatrix}.$$

Here, M^{-1} is basically the classical Dirichlet preconditioner, i.e., a weighted sum of local Schur complements on the interface. In order to obtain a method without sparse direct solvers, we replace the factorizations on the interior part of the subdomains by applications of V-cycles of a sequential algebraic multigrid (AMG). The preconditioner $\widehat{K}^{-1}$ consists of an application of V-cycles of a parallel AMG to $D\widetilde{K}(\tilde{u})$.

Implementation Remarks

We implemented iNL-FETI-DP using PETSc 3.6.2 [10] and hypre 2.10.0b. Our code is written mainly in C/C++. We have used the IBM XL C/C++ compiler for BlueGene, V12.1. We decided to implement the matrix $D\widetilde{K}(\tilde{u})$ and the jump operator B as MPI parallel sparse matrices of the type *MPIAIJ*, which is provided by PETSc. All rows of $D\widetilde{K}(\tilde{u})$ corresponding to the interior and interface nodes of the i-th subdomain are distributed to the same MPI rank, i.e., the local subdomain block $\begin{bmatrix} DK^{(i)}_{BB}(\tilde{u}) & D\widetilde{K}^{(i)}_{B\Pi}(\tilde{u}) \end{bmatrix}$ is assigned to a single MPI rank. The block $D\widetilde{K}^{(i)}_{B\Pi}(\tilde{u})$ couples the local variables to the global variables of the FETI-DP coarse problem. The rows corresponding to the globally assembled FETI-DP coarse space are distributed equally to all MPI ranks, and thus we do not obtain the typical FETI-DP block structure, well known from theory, in our implementation. We always try to distribute a primal variable to one of the MPI ranks handling a neighboring subdomain. This strategy should reduce communication. The rows of B^T are distributed equivalently.

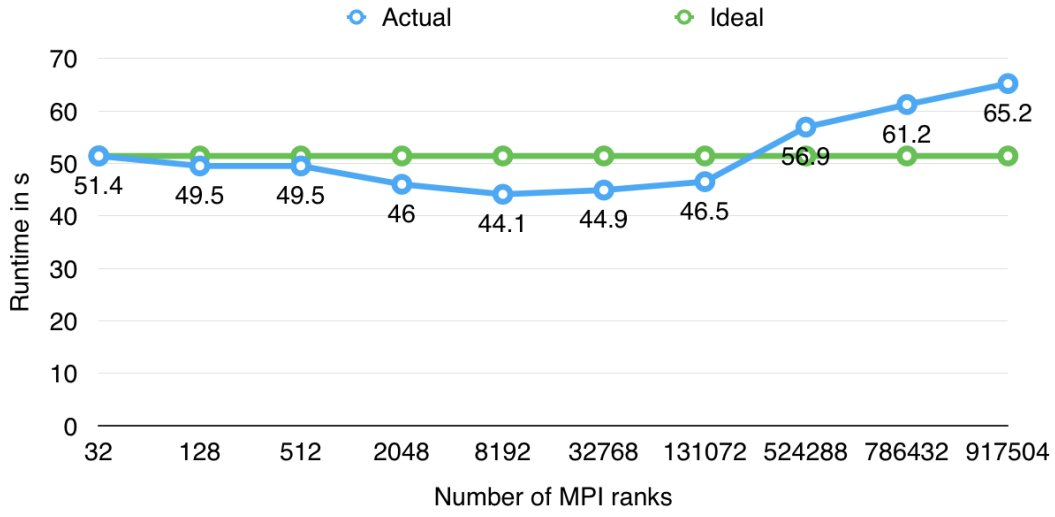
As a preconditioner for $D\widetilde{K}(\tilde{u})$, we always use one V-cycle of BoomerAMG [11]. In the case of difficult elasticity problems, we use the global matrix (GM) approach [12, 13] implemented in BoomerAMG, where the rigid body motions are interpolated exactly. Recently, we have scaled BoomerAMG, using these interpolations adapted for linear elasticity, to half a million MPI ranks [12].

Results

In Table 1, we first present weak scaling results for a linear elasticity problem in two dimensions obtained during the Extreme Scaling Workshop. Here, a rectangular domain is clamped on one of the shorter edges and a volume force is applied in the vertical direction. We refer to this problem as “2D beam”; see also Fig. 1 for the geometry. The final parallel efficiency on the

Table 1: **Linear 2D beam**; one V-cycle of BoomerAMG with nodal HMIS coarsening and GM2 interpolation is used in all cases; **It.** denotes the number of GMRES iterations; the baseline of the parallel efficiency **Eff.** is the time to solution on 32 MPI ranks (1 node); the results up to $524K$ MPI ranks are already published in [4]; **P** denotes the truncation of the AMG interpolation operators; **d.o.f.** denotes the size of the problem; the last two rows have an increased problem size per core and are thus not a continuation of the weak scaling test; the beam has an aspect ratio of 8×1, except 12×1 on $786K$ MPI ranks and 14×1 on $917K$ MPI ranks.

#MPI ranks	d.o.f.	P	It.	Time to Sol.	Eff.	Time Ass. eq. (2)	Time Setup $\hat{K}^{-1}$	Time Setup M^{-1}	Time GMRES
32	1 644 162	-	27	51.4s	100%	5.7s	8.7s	2.9s	32.6s
128	6 565 122	-	25	49.5s	104%	5.8s	8.8s	2.9s	30.4s
512	26 237 442	-	23	49.5s	104%	5.9s	8.9s	2.9s	30.3s
2 048	104 903 682	-	22	46.0s	112%	5.8s	8.9s	3.0s	26.9s
8 192	419 522 562	-	20	44.1s	117%	5.9s	9.1s	3.0s	24.5s
32 768	1 677 905 922	-	20	44.9s	115%	6.2s	9.3s	3.0s	24.6s
131 072	6 711 255 042	-	20	46.5s	111%	6.7s	9.8s	3.0s	24.6s
524 288	26 844 282 882	-	22	56.9s	90%	9.3s	11.1s	3.3s	27.8s
786 432	40 266 383 362	- 4	24 24	64.8s 61.2s	79% 84%	11.7s 11.7s	12.1s 9.7s	3.4s 2.9s	30.7s 29.9s
917 504	46 977 433 602	- 4	22 24	66.0s 65.2s	78% 79%	12.5s 12.5s	12.3s 10.2s	3.6s 3.1s	28.3s 30.3s
917 504	59 455 641 602	4	32	89.8s	–	14.3s	11.8s	3.8s	50.4s
917 504	73 401 856 002	4	27	95.8s	–	16.4s	13.9s	4.5s	50.9s



complete JUQUEEN, using 32 MPI ranks per node, is 79% and thus sufficient. Intermediate runs with efficiencies of more than 100% benefit from a lower number of GMRES iterations, i.e., a numerical effect. In the last two rows of Table 1 the problem size is then increased to up to 73 billion degrees of freedom, which corresponds to 160K degrees of freedom per core. Choosing the inexact reduced nonlinear FETI-DP method, where sparse direct solvers are used on the subdomains, the largest system we solved on 786K cores of Mira BlueGene/Q consisted of 62 billion degrees of freedom; see [14]. This corresponds to roughly 78K degrees of freedom per core, which is half as much as we can handle with iNL-FETI-DP per core.

In Table 2, we then present results for a large nonlinear hyperelasticity problem on a long

Table 2: **Nonlinear 2D beam** of size 12x1; same setup as in Table 1, but nonlinear Neo-Hooke elastic material.

#MPI ranks	d.o.f.	Newton It.	It.	Time to Sol.	Time Ass. eq. (2)	Time Setup $\hat{K}^{-1}$	Time Setup M^{-1}	Time GMRES
786 432	15 729 305 602	5	110	168.5s	29.9s	50.1s	11.3s	60.5s

Table 3: **Three dimensional linear elasticity problem**; investigation of different AMG approaches as preconditioner in iFETI-DP; **U** and **H** refer to the unknown and hybrid AMG approach, respectively; **P** denotes the truncation of the interpolation operator; **thold** denotes the AMG coarsening threshold.

#MPI ranks	d.o.f.	AMG / P / thold	It.	Time to Sol.	Eff.	Time Ass. eq. (2)	Time Setup $\hat{K}^{-1}$	Time Setup M^{-1}	Time GMRES
4 096	50M	U/4/0.3	52	43.9s	100%	12.5s	4.2s	1.1s	25.2s
32 768	405M	U/4/0.3	49	48.6s	90%	13.1s	6.9s	1.2s	26.5s
262 144	3 231M	U/4/0.3	48	63.1s	70%	15.7s	12.3s	1.3s	30.7s
		U/4/0.6	48	160.1s	27%	15.8s	78.7s	1.3s	61.4s
		U/3/0.3	51	63.6s	69%	15.7s	12.0s	1.3s	31.6s
		U/3/0.6	55	214.1s	21%	15.8s	99.2s	1.3s	94.9s
		H/4/0.3	44	50.1s	88%	15.7s	6.7s	2.0s	22.7s
		H/4/0.6	OOM	–	–	–	–	–	–
		H/3/0.3	47	49.5s	89%	15.7s	5.7s	2.0s	23.0s
		H/3/0.6	70	60.5s	73%	15.8s	4.8s	1.9s	35.1s
524 288	6 458M	U/4/0.3	47	73.3s	60%	19.0s	15.6s	1.5s	31.3s
		U/3/0.3	50	72.4s	61%	18.9s	14.0s	1.5s	31.9s
		H/4/0.3	42	56.5s	78%	19.0s	7.6s	2.2s	21.9s
		H/3/0.3	46	56.8s	77%	19.0s	6.7s	2.2s	23.2s

two dimensional beam with aspect ratio of 12×1, using $768K$ MPI ranks.

Finally, in Table 3 we present weak scaling results and AMG parameter studies in three dimensions. We obtain sufficient scalability when considering the best parameters. Regarding the AMG parameters, we obtain large differences in runtimes. In general, hybrid AMG (nodal coarsening and unknown-based interpolation) outperforms the purely unknown-based approach, and a smaller threshold (of 0.3) to identify strong connections leads to a significantly faster convergence.

Acknowledgement

This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 Software for Exascale Computing (SPPEXA) under KL2094/4, RH 122/2. The authors gratefully acknowledge the use of JUQUEEN during the Workshop on Extreme Scaling on JUQUEEN (Jülich, 01.02.2016 - 03.02.2016).

References

[1] Axel Klawonn, Martin Lanser, and Oliver Rheinbach. Towards extremely scalable nonlinear domain decomposition methods for elliptic partial differential equations. 2015. Submitted November 2014. Accepted for publication in SISC. TUBAF Preprint: 2014-13, http://tu-freiberg.de/fakult1/forschung/preprints.

[2] Axel Klawonn, Martin Lanser, and Oliver Rheinbach. EXASTEEL – computational scale bridging using a FE^{2}TI approach with ex_nl/FE^2. Technical Report FZJ-JSC-IB-2015-01,

Jülich Supercomputing Centre, Germany, 2015. JUQUEEN Extreme Scaling Workshop 2015. Dirk Brömmel, Wolfgang Frings and Brian J.N. Wylie (eds.).

[3] Axel Klawonn, Martin Lanser, and Oliver Rheinbach. FE2TI (ex_nl/FE^2) EXASTEEL – bridging scales for multiphase steels, 2015. http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/FE2TI/_node.html.

[4] Axel Klawonn, Martin Lanser, and Oliver Rheinbach. A highly scalable implementation of inexact nonlinear FETI-DP without sparse direct solvers. 2015. Accepted to the Proceedings of the ENUMATH Conference 2015. TUBAF Preprint: 2015-17, http://tu-freiberg.de/fakult1/forschung/preprints.

[5] Martin Lanser. *Nonlinear FETI-DP and BDDC Methods.* PhD thesis, Universität zu Köln, 2015.

[6] Axel Klawonn and Oliver Rheinbach. Inexact FETI-DP methods. *Internat. J. Numer. Methods Engrg.*, 69(2):284–307, 2007.

[7] Axel Klawonn and Oliver Rheinbach. Highly scalable parallel domain decomposition methods with an application to biomechanics. *ZAMM Z. Angew. Math. Mech.*, 90(1):5–32, 2010.

[8] Oliver Rheinbach. Parallel iterative substructuring in structural mechanics. *Arch. Comput. Methods Eng.*, 16(4):425–463, 2009.

[9] A. Klawonn, M. Lanser, and O. Rheinbach. Nonlinear FETI-DP and BDDC methods. *SIAM J. Sci. Comput.*, 36(2):A737–A765, 2014.

[10] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[11] Van E. Henson and Ulrike Meier Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41:155–177, 2002.

[12] Allison H. Baker, Axel Klawonn, Tzanio Kolev, Martin Lanser, Oliver Rheinbach, and Ulrike Meier Yang. Scalability of classical algebraic multigrid for elasticity to half a million parallel tasks. 2015. Submitted 11/2015 to Lect. Notes Comput. Sci. Eng. TUBAF Preprint: 2015-14, http://tu-freiberg.de/fakult1/forschung/preprints.

[13] Allison H. Baker, Tzanio V. Kolev, and Ulrike Meier Yang. Improving algebraic multigrid interpolation operators for linear elasticity problems. *Numer. Linear Algebra Appl.*, 17(2-3):495–517, 2010.

[14] Axel Klawonn, Martin Lanser, and Oliver Rheinbach. FE^{2}TI: Computational scale bridging for dual-phase steels. 2015. Accepted to ParCo 2015. TUBAF Preprint: 2015-12, http://tu-freiberg.de/fakult1/forschung/preprints.

HDF5 import module for the spiking neuronal simulator NEST

Till Schumann and Fabien Delalondre
Blue Brain Project

Description of the Code

The Neural Simulation Tool NEST [1] is a C++ application for simulating large heterogeneous networks of point neurons. It enables the simulation of large-scale neuronal networks on supercomputers. Therefore, NEST distributes the network over the available compute nodes. Each process creates its own part of the network and only stores the synapses to its own neurons. It uses both the Message Passing Interface (MPI) and OpenMP/Pthreads. A large scale data-driven simulation must be integrated into the standard workflow of NEST. The neurorobotics team at the Blue Brain Project (BBP [2]) generated a full point neuron mouse brain model. This model was derived from experiments at the Allen Institute for Brain Science [3] and the reconstruction of the somatosensory cortex from the Blue Brain Project. The circuit of the model contains 16 TB of neuron and synapse information. A new import module for NEST should enable an efficient loading of the whole circuit. Due to differences in the NEST internal data structure and the data delivered by the circuit generation, a transformation of the circuit data is necessary. Synapse information contains pre-synaptic and post-synaptic neuron identifiers in addition of its model parameters. Because of the in vitro injection methods, the synapse information maps the synapse from the pre-synaptic to the post-synaptic neurons. For multi process simulations, NEST distributes all neurons based on a modulo function. Because of memory optimizations the synapses are only stored on the post-synaptic node. This means that the synapse information is stored on the compute node where the post-synaptic neuron is located and hence it is necessary to transform the data. Preprocessing of the input data should be avoided in order to maintain its original format for future changes in the circuit generation. The resulting implementation (see Figure 1) loads the synapse information in parallel, distributes it to the target nodes and stores it in the NEST data structure. The goal of this scaling study is to analyse the performance of the loading of a data-driven simulation inside of NEST. In addition to the usability tests, it will help to identify the bottlenecks of the current implementation.

Multiprocessor simulations with NEST make use of available compute node cores and memory. From experience, a good choice is to use one MPI rank with 16 threads per Blue Gene/Q compute node for simulations with NEST.

Results

The I/O performance of the implementation affects mainly its usability. Runs at different scales should give detailed information about the attained bandwidth. The bandwidth is calculated by dividing the imported number of bytes by the wall-clock time t of the import module:

$$bandwidth = bytes\ per\ synapse * \frac{number\ of\ synapses}{t} [\frac{Bytes}{s}] \quad (1)$$

The bandwidth corresponds to the speed for copying the data from file into the NEST data structure. Since this includes redistributing data in memory, bandwidth will be lower than

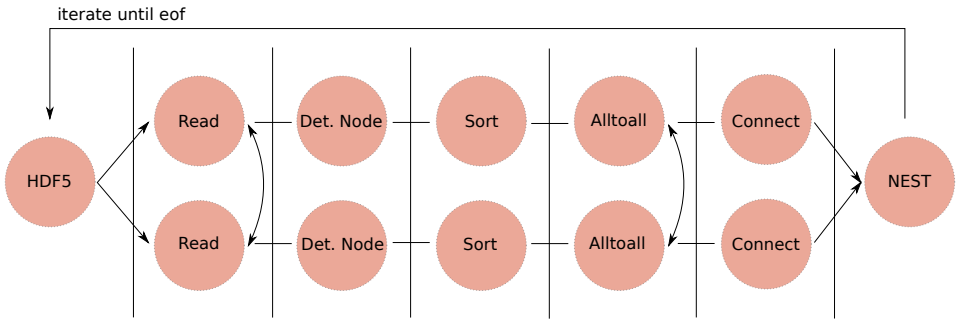


Figure 1: Illustration of the algorithm: The algorithm can be divided into five steps, which are called iteratively. *Read* reads a set of synapses from file into the internal data structure. *Det. Node*, *Sort* and *Alltoall* reorders the synapses on the rank. *Connect* stores the synapses from buffer into the NEST data structure. Steps are repeated until all (or a specified number of) synapses are imported. *Read* and *Alltoall* contain collective MPI operations (all other tasks are executed independently by ranks).

that of the filesystem itself where data is just read from disk. Prior to the workshop, the *connect step* was detected as the bottleneck of the algorithm when running on two racks of IBM Blue Gene/Q. An OpenMP thread-parallelization of the *connect step* achieved a speed-up of the step. Further analysis should highlight other bottlenecks on scales up to 28 racks on IBM Blue Gene/Q. First an optimal block size of read data in one iteration is determined. The theoretical optimal value is equal to the block size of the filesystem. For the JUQUEEN /work filesystem the block size is 4 MB. Several runs using different buffer sizes around 4 MB are performed. The achieved bandwidth (Figure 2) is relatively constant for block sizes between

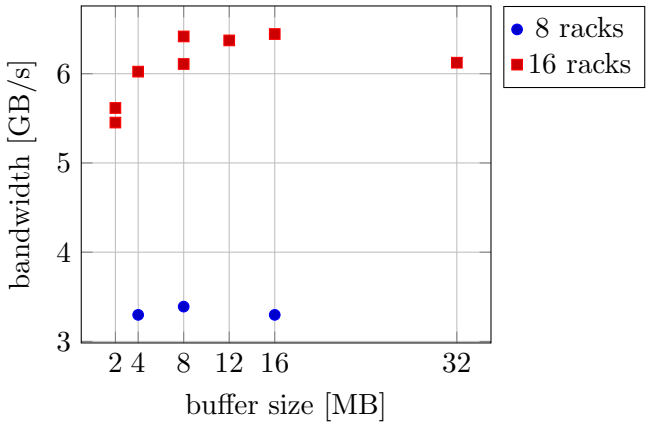


Figure 2: Bandwidth comparison of the import module using different buffer sizes. Multiple runs are preformed with 8 (blue dots) and 16 (red squares) on JUQUEEN.

4 and 30 MB on 8 and 16 racks. As a result, we choose a block size of 12 MB for the following runs.

One strong and two weak scaling scenarios were tested, with properties listed in Table 1.

Figure 3 and 4 plots the bandwidth measured for the strong and weak scaling runs over number of racks, respectively. The achieved bandwidth is approximately 5% percent of the theoretical peak. In order to find the bottleneck, we manually instrumented the code with

Table 1: Properties of scaling runs

name	scaling	importing	rpn	tpp	racks
First	strong	1.9 TB	1	16	1, 2, 4, 8, 16, 28
Second	weak	900 MB/node	1	16	1, 2, 4, 8, 28
Third	weak	220 MB/node	1	16	1, 2, 4, 8, 16, 28

Table 2: Scaling runs of the NEST import module using independent read calls on JUQUEEN. T and *sib* corresponds to the wall-clock time in seconds and number of synapses in buffer of the First, Second and Third scaling runs (Table 1). M equates an factor of 2^{20} for number of synapses. On each rack 1024 nodes are used.

racks	MPI ranks	First T	First *sib*	Second T	Second *sib*	Third T	Third *sib*
1	1024	1222	80640M	523	40320M	123	8960M
2	2048	642	80640M	593	80640M	118	17920M
4	4096	363	80640M	661	161280M	139	35840M
8	8192	325	80640M	827	322560M	245	71680M
16	16384	208	80640M	-	-	364	143360M
28	28672	168	80640M	1401	1128960	497	250880M

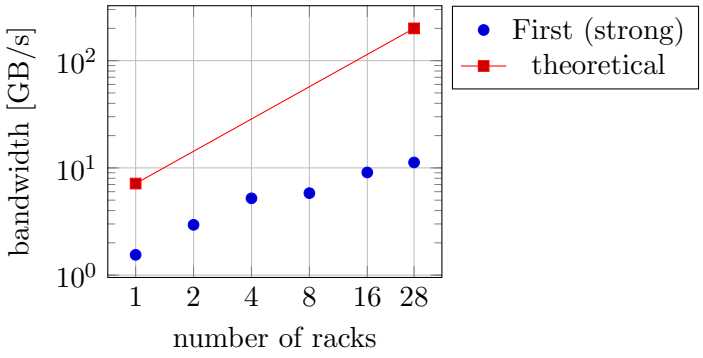


Figure 3: Bandwidth comparison of strong scaling run versus theoretical peak (Table 1). The plotted values are taken from Table 2. The *theoretical* line shows the approximate peak filesystem bandwidth.

Score-P region annotations to get wall-clock timings of each step. Therefore, we look at the proportions of the various steps (see Figure 5).

Apparently, *load* and *alltoall* steps mainly affect the runtime of our module. *load* step performs I/O using the HDF5 library to read synapses from disk into the internal data structure. *alltoall* exchanges synapses between the ranks. Both steps contain collective MPI calls, however, a detailed look at the call stack of the `H5Read` function shown in Score-P profiles indicated that HDF5 was using non-collective individual MPI I/O internally. Using the HDF5 property list interface we identified that the internal data structure of the import module prevented HDF5 from using MPI collective file read operations. Collective read operation could result in better read performance, therefore a comparison of collective versus independent read opera-

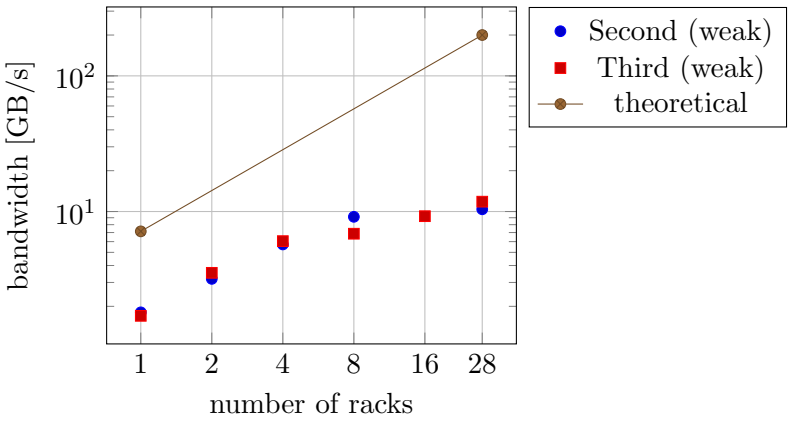


Figure 4: Bandwidth comparison of weak scaling runs versus theoretical peak (Table 1). The plotted values are taken from Table 2. The *theoretical* line shows the approximate peak filesystem bandwidth.

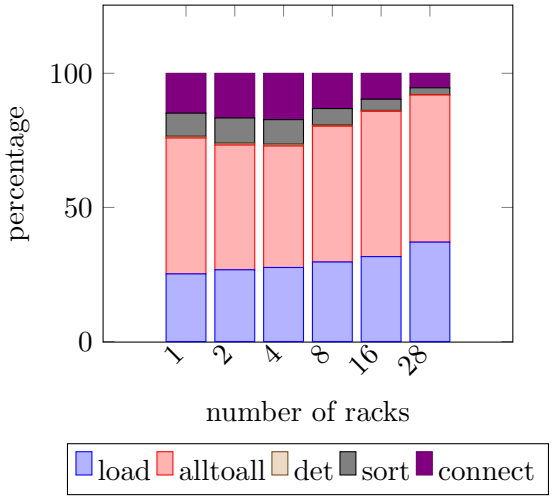


Figure 5: Measured step duration percentages of the total time on different scales. Values are the means of percentages for all nodes, extracted from the First scaling run (Table 1).

tions should show possible benefits. Because of the limited time available during the workshop, we focused on the comparison of a reduced algorithm, which allows an easy change of the used data structure. We use two test implementations, with and without the adapted data structure. The new algorithm reduces import iterations to the *load* part with a following MPI barrier. The MPI barrier should simulate any synchronising collective operations inside of the *alltoall* task. Figure 7 illustrates the benefit of collective versus independent read operations on the JUQUEEN system for 16 racks. Even though the mean time of the read operations is similar, the serializations of the read affects the imbalance and therefore the idle times at the next collective operations. Therefore, a change in the data structure should bring great benefit to our implementation. We are confident that the optimization will result in better usage of the available bandwidth.

Table 3: Scaling runs of test implementation to compare collective versus independent read calls on JUQUEEN. *bytes per synapse* is 24 for all runs. *M* equates an factor of 2^{20} for number of synapses. On each rack 1024 nodes are used.

racks	MPI ranks	MPI-IO	synapses in buffer	t [s]	bandwidth [GB/s]
2	2048	independent	8960M	37	5.7
4	4096	independent	8960M	22	9.7
16	16384	independent	286720M	407	16.5
0.5	512	collective	8960M	71	3.0
2	2048	collective	8960M	29	7.2
4	4096	collective	8960M	13	16.7
8	8192	collective	8960M	8	26.3
16	16384	collective	286720M	191	35.1

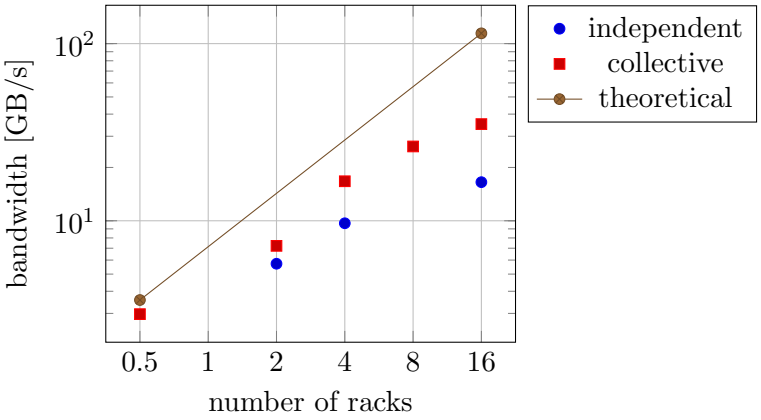


Figure 6: Comparison of achieved bandwidth from MPI independent versus collective read operations. Values from Table 3 are used. Theoretical line shows the limits of the JUQUEEN filesystem.

Conclusion

Even though large scale data-driven simulation are already possible with the developed import module for NEST, further optimization of the algorithm can increase the bandwidth and therefore lower the time needed. Taking into account that large scale data-driven simulations are getting more interest, we are going to focus in the optimization of the import module.

References

[1] Gewaltig, Marc-Oliver and Diesmann, Markus; *NEST (neural simulation tool)*; Scholarpedia **2.4** (2007): 1430;

[2] *Blue Brain Projcet*; `https://bluebrain.epfl.ch`;

[3] *Allen Institute for Brain Science*; `http://alleninstitute.org`;

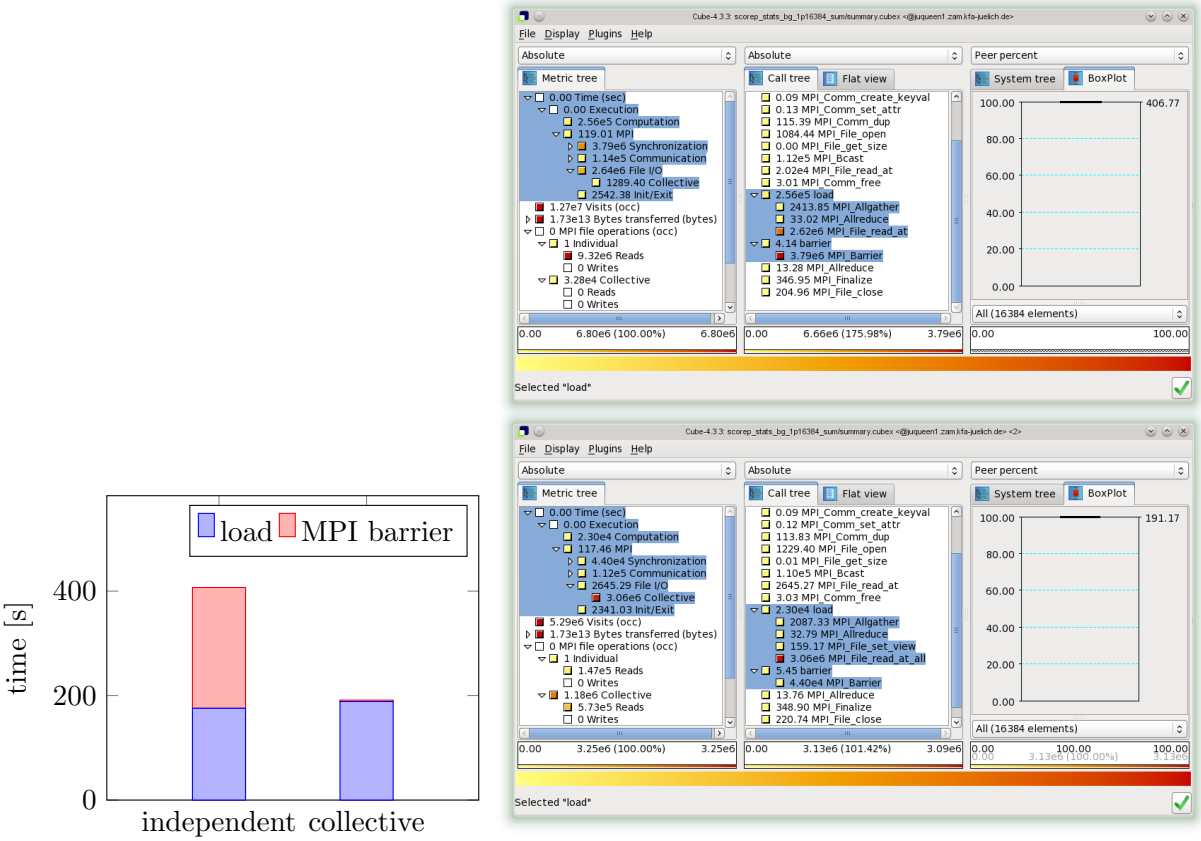


Figure 7: Comparison of load using independent vs. collective MPI file read operations and subsequent barrier synchronisation executed on 16 racks with 16384 MPI ranks. Time values are taken from Score-P profile measurements: (upper) independent using `MPI_File_read_at`, (lower) collective using `MPI_File_read_at_all`.

p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement

Carsten Burstedde and Johannes Holke
INS, University of Bonn

Description of the Code

We examine the scalability of the `p4est` code for parallel adaptive mesh refinement (AMR) [1]. This code implements several algorithms to create a dynamic distributed mesh data structure, to refine, coarsen, and 2:1 balance it (see also [2]), and to repartition it between the parallel processes. Additional algorithms may be called to obtain topological information about the mesh, such as to search or iterate through it [3], or to identify a so-called ghost layer of off-process neighbor elements and transfer data between them. Initial tests of the latter functionality, called ghost exchange, is discussed in this report, together with results of refinement and partitioning.

The basic meshing concept we follow is to divide the domain conformingly into one or more logically hexahedral blocks. One block is suitable for meshing a cube or a torus, and moderate numbers usually suffice to mesh shapes like the spherical shell with good aspect ratio [4]. Complex domains as shown in Figure 2 may be subdivided using mesh generators. This feature is strictly optional, but powerful when needed. (If a mesh generator creates tetrahedra, such as Tetgen [5], we divide each one into four cubes in a preprocessing step.) Each of the coarse blocks becomes an octree by subdividing it arbitrarily into octants. This data structure is fully distributed and dynamic, such that meshes can be modified during runtime.

The parallel arrangement of data is guided by a space filling curve; see Figure 1. This approach allows for fast dynamic repartitioning; see Figure 3 for recent results obtained on the JUQUEEN supercomputer.

The design of `Partition` contains one `MPI_Allgather` call on one integer per rank (or two calls if we use the extra feature to align the elements to allow for coarsening [6]) , $\mathcal{O}(N/P)$ memory traversal and movement, and $\mathcal{O}(1)$ point-to-point messages per rank of total length $\mathcal{O}(N/P)$ with known sender/receiver arrangements. Here, N/P is the number of elements per

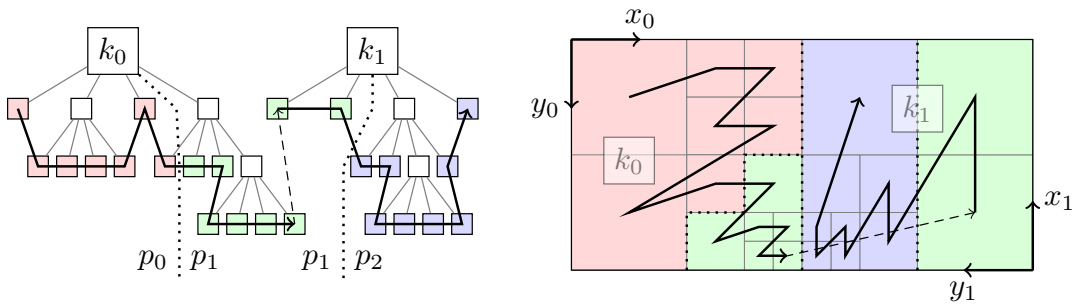


Figure 1: An example 2D mesh of two trees k_1 and k_2. It is partitioned between three processes p_0 through p_2 (color coded). The concept in 3D is analogous.

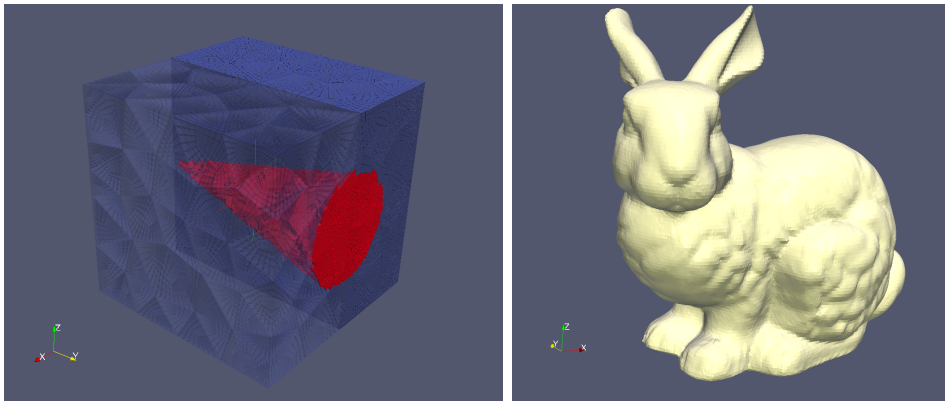


Figure 2: Left: The box mesh used in the 3D tests. Here we show a uniform refinement of level 2 (blue) with adaptive refinement to level 3 (red). Right: The Stanford bunny mesh from the Stanford University Computer Graphics Laboratory [10]. This version of the mesh consists of 495,511 tetrahedra.

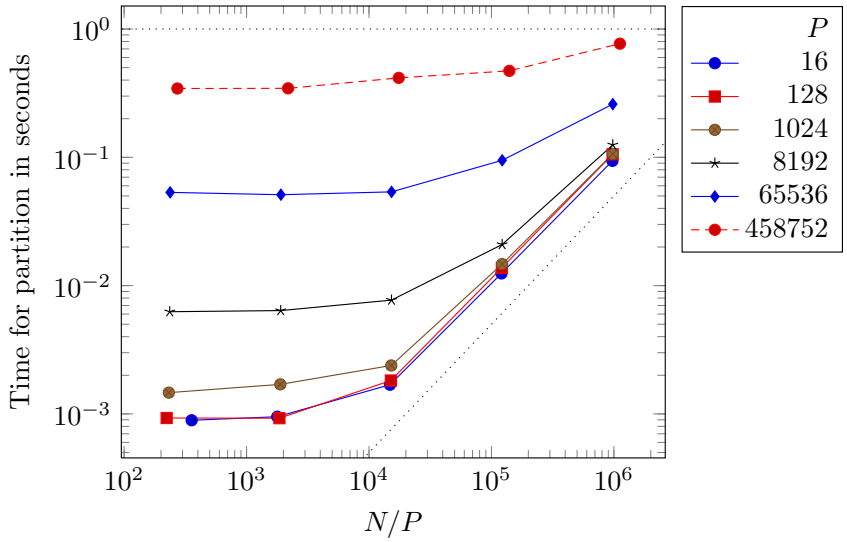


Figure 3: Time of `Partition` plotted against the number of elements N divided by MPI ranks P, on a mesh derived from six trees. Each line corresponds to varying N for a fixed P. All results are in between ideal strong scaling (diagonal line on the bottom right) and an absolute run time of under one second (top horizontal line). The largest run manages over $5 \cdot 10^{11}$ elements on the full size of JUQUEEN.
Strong scalability would be identified by keeping N constant and varying P. In the diagram, that means starting on the left, then moving one point to the right and one line down in each step. Towards the lower right, we approach the plotted diagonal since the lines for different P are on top of each other, indicating near-optimal scaling above 10^4 elements/rank and up to 1024 ranks. Weak scaling can be judged by looking vertically—keeping N/P constant should result in identical runtimes, which is satisfied by the lines with smaller P.
The results indicate that the timings become communication bound, which can be explained by the fact that the `Partition` algorithm has parts whose absolute run time depends on P, not N.

process. Thus, in contrast to say an explicit time step in a PDE solve, it is hard to determine which mechanism is dominant and what the ideal scaling would look like. The main statement that we would like to make is that our partition function is extremely fast in terms of absolute run time: below one second for $0.5 \cdot 10^{12}$ elements on the full size of JUQUEEN.

`p4est` is a portable code written in C using standard MPI. The basic functionality requires MPI version 1.1, with optional MPI file I/O. We link against zlib for compressing VTK output. Saving a mesh using `p4est_save` uses MPI I/O, while we use one file per rank when writing VTK pvtu/xml graphics.

`p4est` is free software and used in many applications, among them finite volume methods [7], higher order finite element [8] and spectral methods [9]. The latter two have been scaled to 1.57 and 3.14 million MPI ranks on Sequoia and Mira, respectively. `p4est` has been the meshing code demonstrated in ACM Gordon Bell Prize finalists in 2008, 2010 and 2012, and the prize winner for 2015 [8].

Results

In managing the mesh metadata, the `p4est` code handles an essential part of the numerical pipeline. The main requirement is that the parallel meshing algorithms do not slow down a simulation, thus we aim for small run times in absolute terms. Even on the biggest meshes, our algorithms require on the order of seconds to run, down to well below one second for realistic examples. Our main focus is thus to establish scalability to the largest possible problem sizes and to verify that the `p4est` algorithms contribute only a negligible fraction to a simulation's run time.

The test configurations

We describe briefly the tests that we planned to run during the Extreme Scaling workshop.

1. Construct a 3D coarse mesh of 4,580 trees from a tetrahedral mesh of a cube-shaped domain consisting of 1,145 tetrahedra. Create a load-balanced uniform refinement of this mesh at a given initial level (`New`) and then perform one adaptive refinement step (`Refine`). In this refinement step we refine those mesh cells that lie in a cone with tip in the middle of one side of the domain and base on the other, see Figure 2. As a last step we load-balance the refined mesh (`Partition`).

2. With the same configuration use the full JUQUEEN system to construct a big mesh of over $9.4 \cdot 10^{11}$ elements. This would be the largest mesh created with `p4est` so far.

3. Do a similar cone refinement pattern with a coarse mesh of $\sim 2 \cdot 10^6$ trees generated from the Stanford bunny mesh (see Figure 2). This mesh has an impractically large tree connectivity that currently has no relevance.

4. In 2D uniformly refine a coarse mesh of 5 trees modelling a Moebius band geometry to a given level, partition the mesh and run the `ghost_exchange` algorithm to exchange data between ghost elements. We used a data size of 4096 bytes per ghost element.

The first three configurations are designed to read the Tetgen [5] file format to preprocess the coarse mesh of octrees. This format is by design non-parallel, thus we opted for reading it on one processor and broadcast it to avoid loading the file system with redundant I/O. Given that the largest coarse mesh we used has under 500k trees, the total run time of reading and broadcasting the mesh was always below 0.1 seconds.

Realization of the tests

The first day of the workshop was used to set up the example applications and a short strong scaling test for test configuration 1. To fine tune the application and input parameters we did several test runs on one JUQUEEN rack using 128 to 32,768 MPI ranks (32 ranks per node).

After this initialization phase we scaled the test configuration 1 to 16 racks (524,288 MPI ranks) during the day and set up scaling runs on up to 24 racks over night (the full system was not available at this point). Results are shown in Table 1. We could run this configuration on the full JUQUEEN system later in the workshop.

On the second day we set up test configurations 2, 3, and 4. The mesh with $9.4 \cdot 10^{11}$ elements could be created successfully on 28 JUQUEEN racks with 32 MPI ranks per node. Results are shown in Table 2. When using smaller numbers of racks the application ran out of memory due to the size of the mesh.

Similar memory limitations were found when testing configuration 3 with the Stanford bunny. The coarse mesh seemed too big to fit into the 16 GiB memory of JUQUEEN nodes.

At the end of the second day and during the third day of the Extreme Scaling workshop we set up configuration 4 for testing our new `ghost_exchange` function. After first tests with smaller data sizes we set up strong and weak scaling runs using 4 kbytes of data per ghost element. During the workshop we ran on up to 16 racks using 32 ranks per node and in the week after the workshop we set up runs on the whole 28 racks.

The results in Table 3 show that the absolute run time of `ghost_exchange` is always below 62ms, even on meshes with $2.1 \cdot 10^{10}$ elements. These times are so small that a standard scaling plot would be dominated by measurement and execution noise.

Our tests with `ghost_exchange` in 3D ran into MPI errors that we will investigate more closely. This is somewhat puzzling since the code is mostly dimension independent.

Further Notes

During the workshop we faced several issues. In the second night several jobs crashed immediately after start, which was observed by other groups as well and seemed to be a transient issue as resubmitted jobs ran as expected.

Reading the Stanford bunny mesh with $2 \cdot 10^6$ trees did not work, since the application ran out of memory due to the size of the coarse mesh. Given more time we would have been able to generate a smaller mesh of the same input data to run our tests.

As described above before running jobs on 16 and more racks we tested our configurations with smaller refinement levels on 1 rack using between 128 and 32k ranks with and without debugging mode enabled (assertions and extra verification).

As a secondary project we would have liked to test to what extent the MPI-3 shared memory features can save memory when running more than one MPI process per node. We set up a small test program to create a shared memory array and measure memory usage. However, since the shared memory required by this particular test is on the order of 10^2 bytes and its measurement (using `Kernel_GetMemorySize`) displays the used memory on a scale of 10^4 bytes we could not obtain useful results. Due to a tight schedule we did not run more tests with other memory sizes, but we plan to further investigate the shared memory features in the future.

Table 1: Strong scaling run time results for the 4,580-tree box mesh from Figure 2 (left). We generate a distributed uniform level 8 mesh (`New`), refine once more according to the given cone shape (`Refine`) and load-balance (`Partition`). The mesh sizes are $7.68 \cdot 10^{10}$ elements before the final refinement and $1.18 \cdot 10^{11}$ afterwards.

Racks	MPI ranks	`New`	`Refine`	`Partition`
8	262,144	0.486s	2.25s	3.18s
14	458,752	0.722s	1.35s	3.28s
16	524,288	0.802s	1.12s	2.69s
20	655,360	0.980s	1.01s	3.11s
24	786,432	1.158s	0.89s	2.91s
28	917,504	1.366s	0.82s	2.82s

Table 2: We manage a mesh of over $9.4 \cdot 10^{11}$ elements. This mesh is created from 4,580 trees, first refined uniformly to level 9 and then refined once adaptively.

Racks	MPI ranks	# mesh Elements	`New`	`Refine`	`Partition`
28	917,504	940,642,225,005	1.64s	5.60s	14.2s

Table 3: Run time results for `ghost_exchange`. 'lvl' refers to the uniform refinement level of the mesh used. The total number of mesh elements is $5 \times 4^{\text{lvl}}$, ranging between $3.4 \cdot 10^8$ (level 13) to $2.1 \cdot 10^{10}$ (level 16).

racks	MPI ranks	`Exchange`	lvl
4	131,072	9.4ms	13
8	262,144	23.6ms	13
16	524,288	20.4ms	13
4	131,072	15.0ms	14
8	262,144	16.0ms	14
16	524,288	38.2ms	14
28	917,504	34.9ms	14
4	131,072	30.2ms	15
8	262,144	29.2ms	15
16	524,288	42.4ms	15
28	917,504	53.9ms	15
28	917,504	61.4ms	16

Conclusions

This workshop provided us with the opportunity to generate and publish latest results on scalability. While effective development of new features within the code was not possible given the fixed schedule of submitting jobs, we were able to obtain new information on routines and configurations that are not usually covered by the production usage of the code.

We have executed the functions `New`, `Refine`, `Partition`, and `ghost_exchange` implemented by the `p4est` AMR code. We have worked with coarse meshes on the order of 5k trees and created, refined, and partitioned meshes to sizes between 100 and 940 billion elements. Absolute run times of all meshing operations are between a few milliseconds and several seconds depending on the configuration.

References

[1] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[2] Tobin Isaac, Carsten Burstedde, and Omar Ghattas. Low-cost parallel algorithms for 2:1 octree balance. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2012. `http://dx.doi.org/10.1109/IPDPS.2012.47`.

[3] Tobin Isaac, Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing*, 37(5):C497–C531, 2015.

[4] Carsten Burstedde, Georg Stadler, Laura Alisic, Lucas C. Wilcox, Eh Tan, Michael Gurnis, and Omar Ghattas. Large-scale adaptive mantle convection simulation. *Geophysical Journal International*, 192(3):889–906, 2013.

[5] Hang Si. *TetGen—A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator*. Weierstrass Institute for Applied Analysis and Stochastics, Berlin, 2006.

[6] Hari Sundar, George Biros, Carsten Burstedde, Johann Rudi, Omar Ghattas, and Georg Stadler. Parallel geometric-algebraic multigrid on unstructured forests of octrees. In *SC12: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, 2012. ACM/IEEE.

[7] Carsten Burstedde, Donna Calhoun, Kyle T. Mandli, and Andy R. Terrel. Forestclaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws. In Michael Bader, Arndt Bode, Hans-Joachim Bungartz, Michael Gerndt, Gerhard R. Joubert, and Frans Peters, editors, *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 253 – 262. IOS Press, March 2014.

[8] Johann Rudi, A. Cristiano I. Malossi, Tobin Isaac, Georg Stadler, Michael Gurnis, Peter W.J. Staar, Yves Ineichen, Costas Bekas, Alessandro Curioni, and Omar Ghattas. An extreme-scale implicit solver for complex PDEs: highly heterogeneous flow in earth's mantle. In *Proceedings of the SC15 International Conference for High Performance Computing, Networking, Storage and Analysis*, article 5. ACM, 2015.

[9] Andreas Müller, Michal A. Kopera, Simone Marras, Lucas C. Wilcox, Tobin Isaac, and Francis X. Giraldo. Strong scaling for numerical weather prediction at petascale with the atmospheric model NUMA. http://arxiv.org/abs/1511.01561, 2015.

[10] The Stanford University Computer Graphics Laboratory. Stanford bunny dataset, 1994. `http://graphics.stanford.edu/data/3Dscanrep/`, last accessed Feb 16, 2016.

[11] James R. Stewart and H. Carter Edwards. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elements in Analysis and Design*, 40(12):1599–1617, 2004.

PFLOTRAN grain-scale simulations of oxygen intrusion and radionuclide sorption in a heterogeneous rock sample

Hedieh Ebrahimi[1], Paolo Trinchero[1], Jorge Molinero[1], Guido Deissmann[2], Dirk Bosbach[2], Glenn Hammond[3], and Peter Lichtner[4]
[1]AMPHOS 21 Consulting S.L., Passeig de Garcia i Faria, 49-51, 1-1, 08019 Barcelona, Spain
[2]Institute for Energy and Climate Research: Nuclear Waste Management and Reactor Safety (IEK-6) and JARA-HPC, Forschungszentrum Jülich GmbH, 52425 Jülich, Germany
[3]Applied Systems Analysis and Research, Sandia National Laboratories, Albuquerque, New Mexico, USA
[4]OFM Research

Description of the Code

PFLOTRAN is an open source, state-of-the-art massively parallel subsurface flow and reactive transport code. PFLOTRAN solves a system of generally nonlinear partial differential equations describing multiphase, multicomponent and multiscale reactive flow and transport in porous materials. The code is designed to run on massively parallel computing architectures as well as workstations and laptops. Parallelization is achieved through domain decomposition using the MPI-based PETSc (Portable Extensible Toolkit for Scientific Computation) libraries. PFLOTRAN has been developed from the ground up for parallel scalability and has been run on up to 2^{18} processor cores with problem sizes up to 2 billion degrees of freedom [1]. PFLOTRAN is written in object oriented, free formatted Fortran 2003. PFLOTRAN uses parallel HDF5 for file Input/Output [2,3].
The code has the capability to simulate the following processes:

- Single phase variably-saturated flow through Richard's Equation
- Thermo-Hydro processes
- Multiphase Air-Water-Energy
- Surface Flow
- Discrete Fracture Network
- Aqueous Complexation
- Sorption
- Mineral Precipitation and Dissolution
- Multiple Continuum for Heat
- Subsurface Flow-Reactive Transport Coupling
- Multiphase Ice-Water-Vapor Flow
- Structured and Unstructured Grids
- Multiple Realizations
- Multiple Inputs
- Parallel I/O

Problem Description

The problem addressed in the scalability tests with PFLOTRAN is the simulation of the diffusive penetration of oxidative water and radionuclides into an initially anoxic and radionuclide-free rock sample. Mineralogical heterogeneity, which is related to both the buffering capacity of the rock sample and the availability of sorption sites, is described at the pore scale using an extremely fine model discretization (8M cells up to 50M cells) for a rock sample with the model dimension of $14.8 \times 14.8 \times 14.8\text{mm}^3$. The chemical problem mimics the ingress of oxygenated water from one of the six faces of the cube into an initially anoxic environment. Fickian diffusion is the main transport driver for oxygen, which can be consumed by abiotic reactions, such as the non-oxidative dissolution of biotite that provides an input of dissolved Fe(II) ions into the system, and the homogeneous oxidation of dissolved ferrous ions. Both biotite dissolution and ferrous ion oxidation are described as kinetically controlled reactions. Moreover, cesium that can be sorbed by cation exchange at the surfaces of the heterogeneously distributed biotite in the rock volume is added to the boundary water in trace concentration. The objective of the simulated problem is to understand the mutual interplay between pore-scale mineralogical heterogeneity and mineral dissolution and its influence on radionuclide retention during a glacial period. Figure 1 shows a 3D visualization of the rock sample along with mineral distributions.

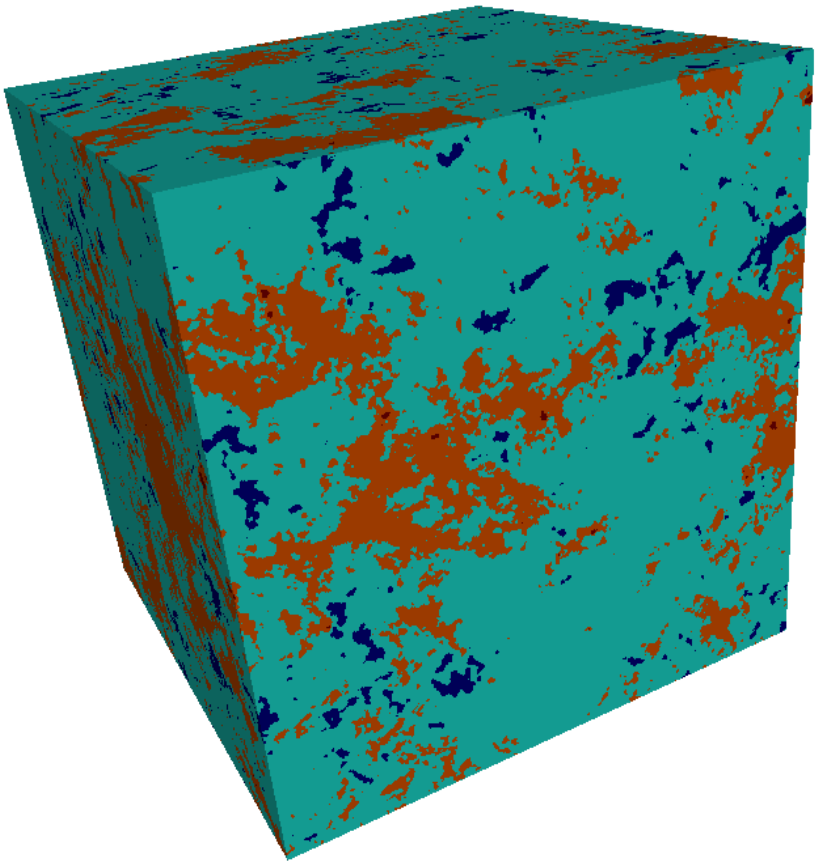


Figure 1: 3D visualization of the Granodiorite rock sample [2]. It includes pores (dark blue) with porosity set to 1 , altered feldspar (cyan) and altered mica (orange) with porosity set to 10^{-2}. The dimensions of the sample are $14.8 \times 14.8 \times 14.8$ mm^3.

Scaling Tests performed during Workshop

During the workshop three sets of tests were carried out using three different model sizes: 8M, ∼16M and ∼50M grid cells models.

The execution of PFLOTRAN is divided into four different steps including initialization, flow, transport and output. The initialization step mainly includes subroutines that read simulation data from HDF5 input files. There were known issues regarding Input/Output, which have been addressed in a later section. Due to scaling issues regarding input/output, writing all output including HDF5 outputs and checkpoints were disabled during the scaling tests and only for the sake of comparison, a few tests were performed with HDF5 output enabled. The models tested have relatively limited degrees of freedom (DOFs). A rule of thumb is that scalability degrades from ideal if the DOF per process is lower than 10k.

Given the limited dimensions of the aforementioned models, a bigger model consisting of 300M grid cells was also prepared. This model was intended to be used for testing PFLOTRAN over multiple racks. However given its dimensions, this model required using the version of PFLOTRAN with 64-bit integers. Unfortunately, given the limited time of the workshop, it was not possible to compile binary file for PFLOTRAN 64-bit as the PFLOTRAN source needed additional code adaptation.

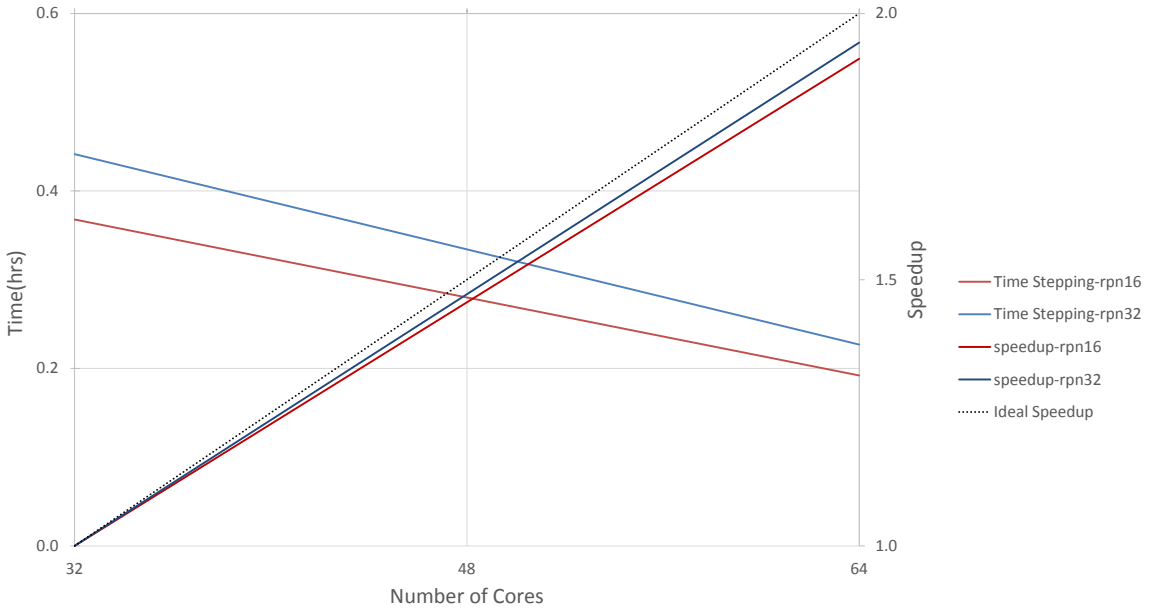


Figure 2: WallClockTime (in hours) and Speedup for Copper Leaching scenario ($32 \times 32 \times 4$ = 4M cell grid); Total DOF=49k. Comparison of 16 and 32 ranks per node.

Nodes	Ranks-per-node	Processes	calc Time [hrs]	init Time [hrs]
1	32	32	0.441	4.42E-04
2	16	32	0.368	3.68E-04
2	32	64	0.227	2.27E-04
4	16	64	0.192	1.92E-04

Figure 2 shows that 32 ranks-per-node (RPN) shows higher relative speedup compared to RPN 16. The difference is more obvious when going to higher degrees of freedom.

16 ranks per node was selected for scaling tests during the workshop as it was observed from preliminary scalability tests on JUQUEEN that it results in faster time stepping. Figure 2 shows the results of scalability tests done for the so-called Copper Leaching problem (a standard regression test of PFLOTRAN). The copper leaching problem consists of a cubic $16 \times 16 \times 16$ m^3 domain with a $32 \times 32 \times 4$ (4M cell) grid and twelve chemical degrees of freedom representing 12 primary basis species. The total number of DOFs is 49152 (∼49k). The total degree of freedom is defined by the number of grid cells times the number of unknowns per grid cells. Another important concept to remember is degree of freedom per process which is defined for each simulation by dividing the total degree of freedom by the number of processes used for that simulation. Figure 2 shows the scalability test results and the Initialization and Time Stepping times for the copper leaching problem ran with 32 and 64 MPI ranks and using RPN 16 and RPN 32. Output has been disabled during the tests.

Three different models were tested during the workshop. For each model, a brief description is given. The three models have the same dimension ($14.8 \times 14.8 \times 14.8$ mm^3) and the same chemical reactions are considered. The difference between the models are the result of different discretizations used and distinct material distribution applied which result from stochastic realizations done.

Model 1: 8M Cells

The first model tested consists of 200 x 200 x 200 grid cells and twenty chemical degrees of freedom representing 20 primary basis species resulting in a total degree of freedom of 160M.

Figure 3 shows the scalability test results for model 1 in red. It scales reasonably up to 1 rack. Time stepping and initialization times are shown in Figure 4 (upper left). The initialization time stays roughly the same and is 0.02 hours. As can be seen from the Table 1, DOF per process (DOF/process) for the three tests is of order of 10k. Also, as can be seen from Figure 4 (upper right), the number of transport linear solver iteration stays roughly linear as the number of cores increases.

Model 2: ∼16M Cells

The second model tested has $251 \times 251 \times 251$ grid cells (15,813,251 cells) and twenty chemical degrees of freedom representing 20 primary basis species which bring about a total degree of freedom of 316,265,020 (316M).

Table 2 shows that DOF per process (DOF/process) for the tests done up to 2 racks is of order of 10k and for 4 and 8 racks reduces below that. As can been seen from Figure 3 in green, Model 2 scales well up to 1 rack and then diverges from ideal speed up. Looking at Figure 4 (middle left), the initialization time stays roughly the same and is 0.03 hours. The effect of the divergence from the ideal speedup is apparent on the gradient of the linear solver iterations in Figure 4 (middle right).

Model 3: ∼50M Cells

The third model tested has $368 \times 368 \times 368$ grid cells (49,836,032 cells) and twenty chemical degrees of freedom representing 20 primary basis species which give a total degree of freedom of 996,720,640 (996M).

As can been seen from the blue line in Figure 3, Model 3 scales well up to 2 racks and then diverges from the ideal speed up line. Looking at Figure 4 (lower left), the initialization time stays roughly the same and is 0.10 hours. The transport linear solver iterations for Model 3 is shown in Figure 4 (lower right). The calculation times and initialization times for Model 3 are shown in Table 3.

Table 1: Strong scaling tests; 8M model (nX=nY=nZ=200); Total DOF=1.60E+8.

Nodes	Processes	DOF/process	calc Time [hrs]	init Time [hrs]	Speedup	Ideal Speedup
256	4096	39062.50	1.005	0.018	1.000	1
512	8192	19531.25	0.480	0.021	2.095	2
1024	16384	9765.62	0.274	0.022	3.672	4

Table 2: Strong scaling tests; 16M model (nX=nY=nZ=251); Total DOF=3.16E+8.

Nodes	Processes	DOF/process	calc Time [hrs]	init Time [hrs]	Speedup	Ideal Speedup
256	4096	77213.13	0.877	0.011	1.000	1
512	8192	38606.56	0.539	0.034	1.626	2
1024	16384	19303.28	0.417	0.034	2.102	4
2048	32768	9651.64	0.184	0.035	4.771	8
4096	65536	4825.82	0.157	0.035	5.583	16
8192	131072	2412.91	0.054	0.035	16.117	32

Table 3: Strong scaling tests; 50M model (nX=nY=nZ=368); Total DOF=9.97E+8.

Nodes	Processes	DOF/process	calc Time [hrs]	init Time [hrs]	Speedup	Ideal Speedup
256	4096	243340.00	0.420	0.039	1.000	1
512	8192	121670.00	0.255	0.097	1.647	2
1024	16384	60835.00	0.132	0.101	3.195	4
2048	32768	30417.50	0.070	0.101	6.031	8
4096	65536	15208.75	0.052	0.099	8.107	16
8192	131072	7604.38	0.025	0.098	16.684	32

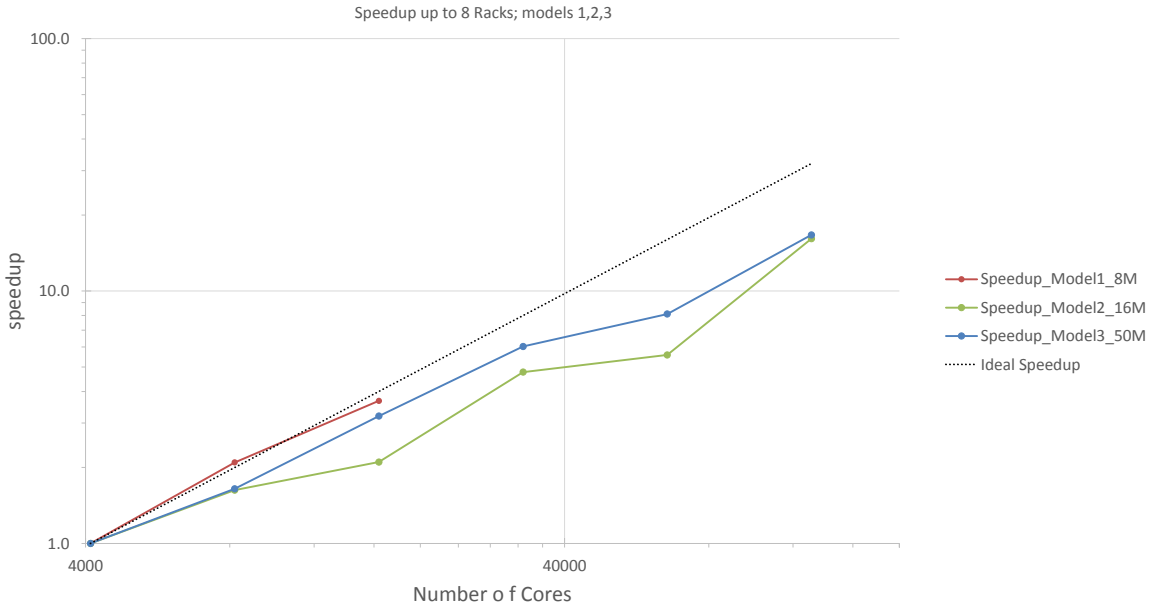
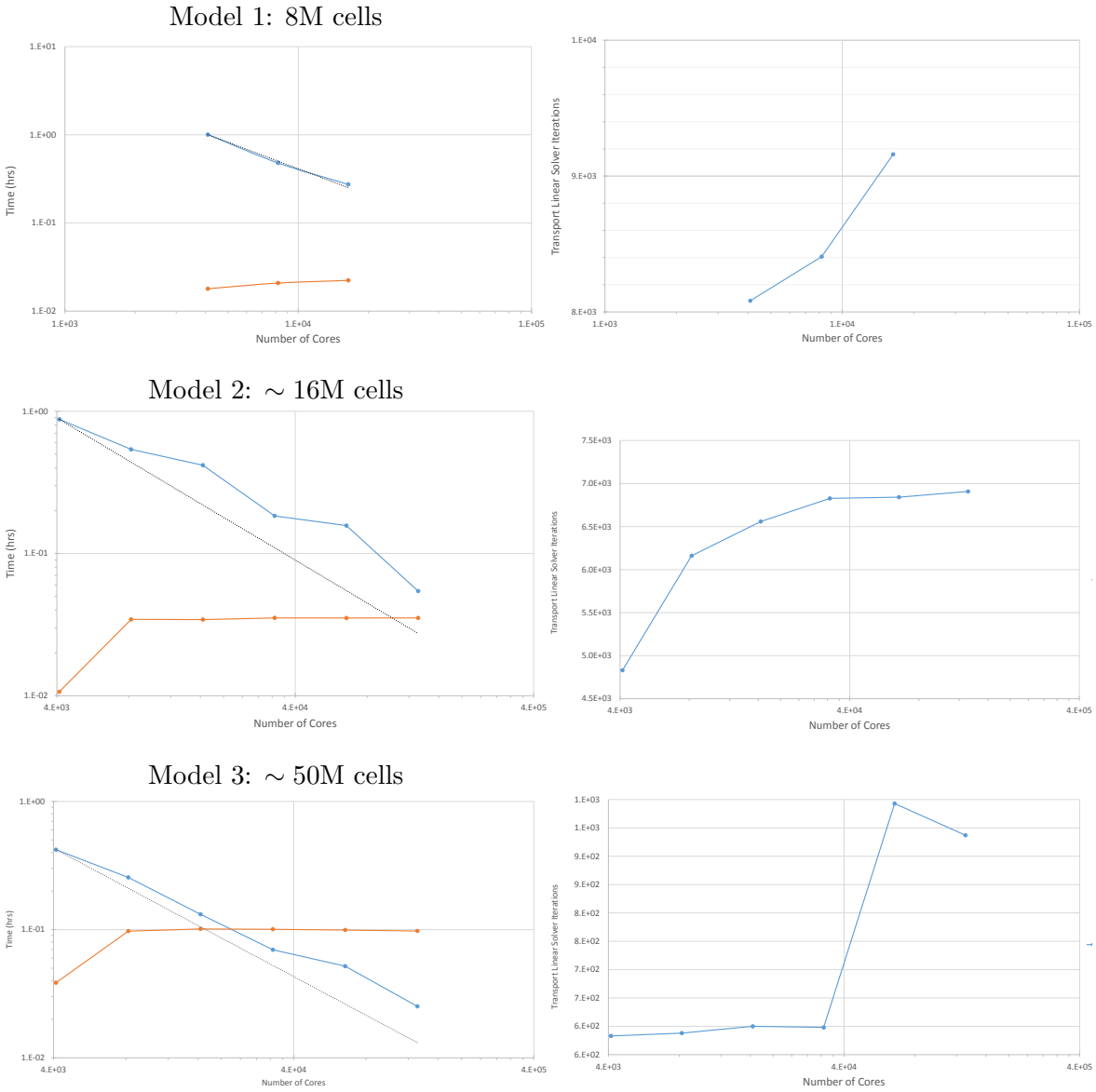


Figure 3: Scalability up to 8 Racks; comparison of 3 models.



(a) Time-stepping (blue) & Initialization (orange). (b) Transport Linear Solver iterations.

Figure 4: Strong scaling comparative analysis of the three models.

Tables 1, 2 & 3 show that the average initialization time for each of the three models approximately stays the same as the number of processes increases. The average initialization time for Model 1 is 0.02 hours or 1.2 minutes whereas for Model 2 it is 0.03 hours or 1.85 minutes, and for Model 3 0.09 hours or 5.34 minutes. The average initialization time increases as the grid discretization becomes finer.

Encountered Problems

HDF5 I/O: PFLOTRAN can output data using the following procedures:

1. Binary HDF5 files may be written collectively from all processes. (Recommended for <10k processes.)

2. A two stage I/O operation using SCORPIO [4] where processes are divided into groups and one process from each group aggregates data from all processes within the group. This helps to minimize congestion when going to >10k processes.

Table 4: Output times.

Nodes	Processes	Output time [sec]
256	4096	584
512	8192	2070
1024	16384	2320

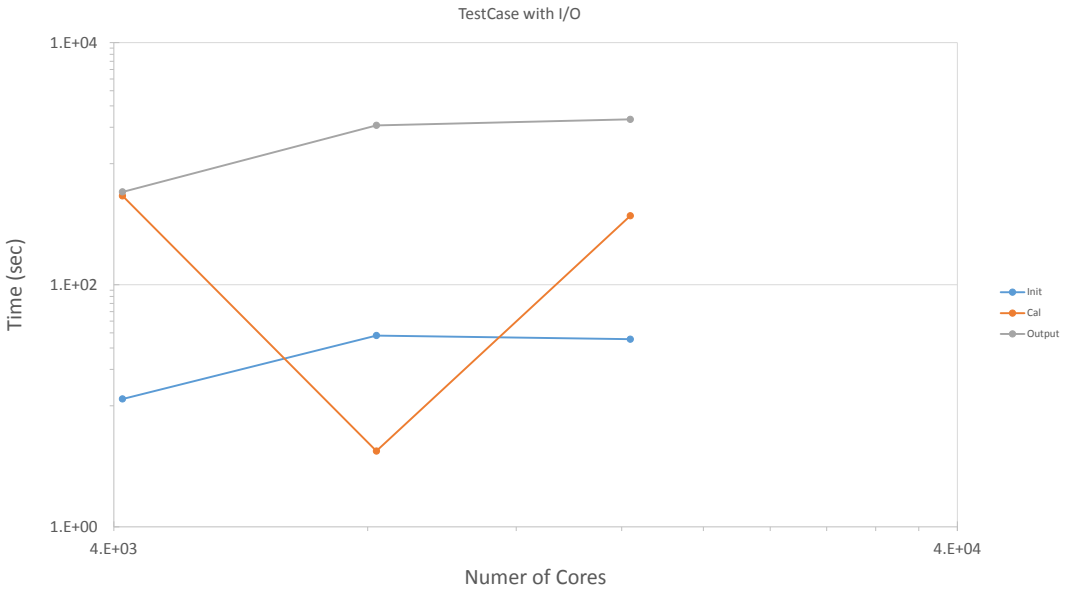


Figure 5: TestCase with file I/O.

Tests up to one rack were done using the first scenario (with file size of 1.5 GB, writing a maximum of two files). Table 4 shows the output times. It is evident that the write times are drastically increasing and are not optimal. As can be seen from Figure 5 write times increase drastically as the number of processes increase which as a result impedes scaling tests using HDF5 I/O.

The solver behavior over multiple racks: PFLOTRAN scales well up to 1 rack. Beyond one rack, the performance degrades significantly. The issue is heavily dependent on breakdown in Krylov solver performance (i.e. the preconditioners break down).

Examining PFLOTRAN file I/O behavior with Darshan: PFLOTRAN was linked with Darshan in order to investigate the file input/output behavior of PFLOTRAN on JUQUEEN. Unfortunately we did not receive any output from Darshan.

PFLOTRAN memory size per process: PFLOTRAN does not provide memory size per process and thus we have not been able to determine this during the workshop.

Conclusion

The three models were found to scale well while the degrees of freedem per process is at least of order of 10k. When this ratio decreases, the scalability is poorer.

Workshop participation has been extremely useful as a feedback to the developers. The code is under continuous development and the results from scaling tests have provided useful insights into existing challenges.

References

[1] `http://www.pflotran.org/docs/user_manual.pdf`

[2] Voutilainen, M., et al. *Pore-space characterization of an altered tonalite by X-ray computed microtomography and the 14C-labeled-polymethylmethacrylate method*, Journal of Geophysical Research: Solid Earth 117.B1 (2012).

[3] `http://www.pflotran.org/`

[4] Sreepathi, Sarat, et al.. *A scalable two-phase parallel I/O library with application to a large scale subsurface simulator*, 20th International Conference on High Performance Computing, IEEE, 2013.

[5] Hammond, G.E., P.C. Lichtner and R.T. Mills. *Evaluating the Performance of Parallel Subsurface Simulators: An Illustrative Example with PFLOTRAN*, Water Resources Research **50**, 2014. [DOI: 10.1002/2012WR013483.]

SLH Seven-League Hydro Code

Philipp V. F. Edelmann and Friedrich K. Röpke
Heidelberger Institut für Theoretische Studien, Schloss-Wolfsbrunnenweg 35,
69118 Heidelberg, Germany

Description of the Code

The Seven-League Hydro (SLH) code is an astrophysical hydrodynamics code with a focus on applications in stellar astrophysics. Its distinguishing features are special low Mach number discretizations of the hydrodynamical fluxes and implicit time stepping. Both enable us to cover the long time scales involved in many problems of stellar evolution.

As the code uses implicit time discretization, we need to solve a nonlinear system of equations at every time step. We use the Newton-Raphson method for this, which involves the solution of a linear system for each Newton iteration. This system is extremely large. For a grid size of 1024^3 cells the number of unknowns is about 5 billion. We use iterative linear solvers, mostly BiCGSTAB, GMRES, and multigrid, for this system. This makes up the main workload in most cases. As the Mach numbers in our simulations are typically $\lesssim 10^{-3}$, it is still more efficient than explicit methods due to the much larger time steps that are possible with an implicit method.

Conventional compressible solvers are known to yield wrong results if used for flows at low Mach numbers. This problem is often fixed by modifying the underlying equation (e.g. Boussinesq or anelastic approximation). This causes problems for flows that also include regions that are not strictly in the low Mach regime. Therefore, SLH takes the approach to modify not the equations, but the numerical solver, such that it behaves correctly also in the low Mach number limit. The method of choice in SLH is a flux preconditioned Roe solver [1]. This makes SLH essentially an all Mach number code.

SLH is a relatively new code (development started in 2009) written in Fortran 95. We use MPI parallelization with an optional hybrid mode using OpenMP. It does not rely on external libraries except for an implementation of LAPACK. In particular, the implementations of the linear solvers are tailored to the sparse matrix structure that occurs in the solution of the Euler equations.

Output is written in a custom binary format. Typically one file per time step is used, written using collective MPI-IO calls. There is another mode which writes one file per MPI process.

As an example application Figure 1 shows a volume rendering of a simulation of the surface of a white dwarf star in the phase leading to a classical nova explosion.

Results

During the workshop we were able to run our test setup on all 28 racks of JUQUEEN. Previous tests were only run on up to 8 racks during normal operation. Also we were able to test our I/O routines on up to 16 racks.

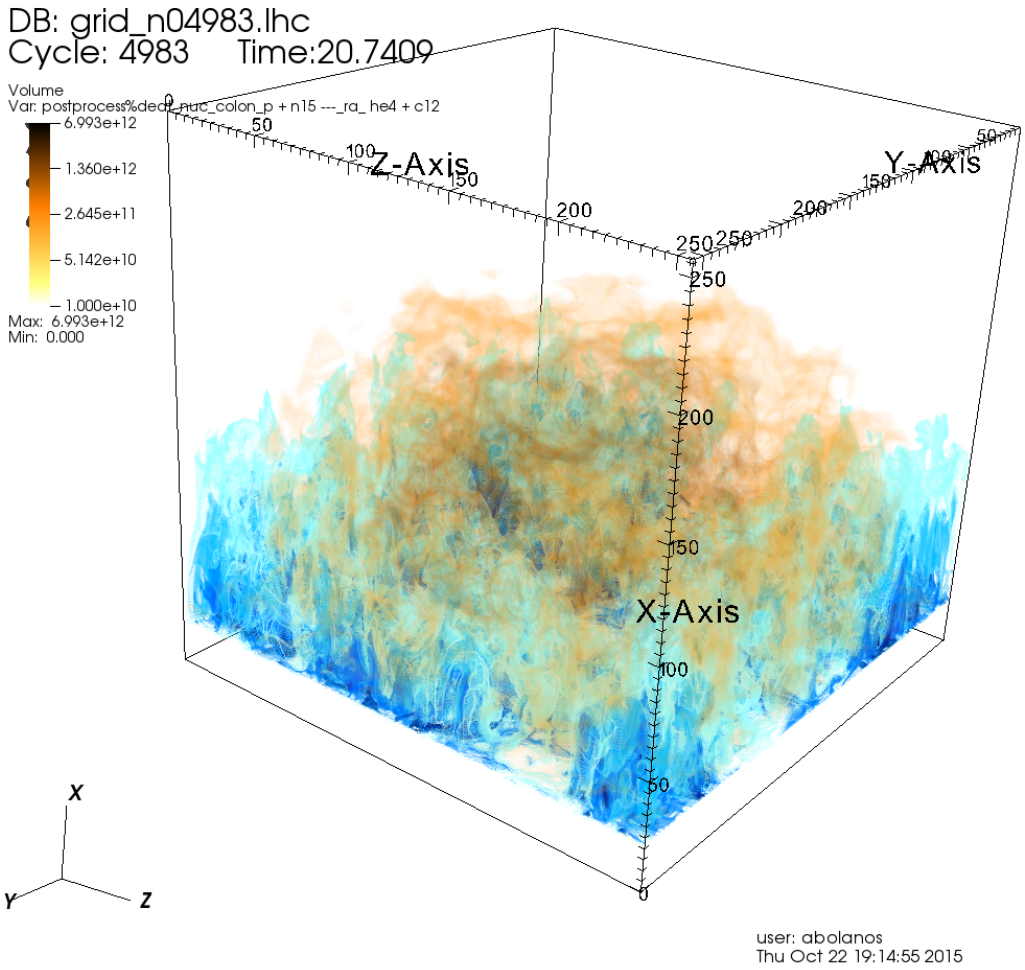


Figure 1: Nuclear reactions in the accreted envelope of a white dwarf star, a possible progenitor to a classical nova. The blue color shows the contribution of the reaction $^{12}\mathrm{C}(\mathrm{p},\gamma)^{13}\mathrm{N}$ to the energy release, the orange color shows that of $^{15}\mathrm{N}(\mathrm{p},\alpha)^{12}\mathrm{C}$. The simulation was run on JUQUEEN as part of project HWB07. Image credit: Alejandro Bolaños (Würzburg University)

The problem setup for the test runs was the Taylor–Green vortex [2, 3], a decaying vortex initial condition, which is often used to study turbulence properties of fluid dynamics schemes. As a part of project HWB07 we previously ran this setup for a range of resolutions (up to 1024^3) and discretizations. We analyzed the turbulent energy spectrum and the numerical Reynolds number, especially its behavior at low Mach numbers.

There were no unexpected problems with SLH running in a number of different configurations on JUQUEEN. The fact that SLH uses a static domain decomposition proved to be a small nuisance when scaling to the full machine as the grid size needs to accommodate the number of processes. This could be fixed by allowing a nonuniform distribution of the grid to the MPI ranks but as long as SLH does not include adaptive mesh refinement and load balancing this would likely deteriorate the scaling properties. An alternative approach would be to use less than 16 processes per node. The efficiency of the OpenMP part of the parallelization should be improved in this case. On JUQUEEN SLH currently only uses OpenMP for running 4 threads on the same core as this was previously determined to be the most efficient configuration.

We performed two main series of scaling tests, one on a 1920^3 grid ranging from 4 to 24 racks and another one on a 2688^3 grid ranging from 14 to 28 racks. The results are shown in Fig. 2 and 3. As it is almost purely a local problem with mostly constant compute time, the computation of the fluxes (FS) always scales almost ideally. The overall scaling behavior is mostly dominated by the linear solver (LS) component, which takes roughly 90% of the total computation time. The fact that scaling is more than ideal for the first step in the runs with a 1920^3 grid is probably due to the fact that the configuration with the fewest number of nodes used about 0.9 GiB/core, almost all available memory. The deterioration of scaling efficiency at the last data point is possibly due to the non ideal domain decomposition of $192 \times 64 \times 32$ instead of $96 \times 64 \times 64$. The runs with a 2688^3 grid show a promising scaling behavior, reaching 88% of the ideal speed-up.

All the above tests were run without I/O but we performed a separate set of I/O benchmarks by writing the typical output of a 1920^3 grid (about 264 GiB). We tested writing to one large file using MPI-IO and writing to one file per process using standard functions from C stdio. No file system hints were set during the tests. The files were created prior to the measurement, which had a positive impact on performance. The results are shown in Table 3. Writing one file per process generally delivers better performance but it would have to be accompanied by a post-processing step that aggregates the output into one file for practical reasons. Proper use of file system hints could possibly make the *one file* scenario significantly faster.

Apart from showing that SLH can scale reasonably to the full machine, participation in the scaling workshop enabled us to learn about new analysis tools and platform-specific settings, especially with respect to I/O. Additionally, we were able to get advice directly from one of the SIONlib developers, which helped us evaluate its usefulness for our I/O scenario and devise a strategy of integrating it with SLH with minimal programming effort.

Conclusions

The tests during the scaling workshop showed that SLH can scale to the full machine (Fig. 3) at 88% of the ideal speed-up compared to half the machine. Most potential for future improvement is definitely in the linear solver part of the code, which includes most of the collective communication. The I/O tests revealed the fact that SLH achieves better performance when writing one file per process even for more than 10^5 processes provided that the files exist in advance. This prompted the decision to include SIONlib as an additional output method for SLH. We will also investigate the impact of file system hints for MPI-IO output in future tests.

Acknowledgments

This work was supported by the Klaus Tschira Foundation.

References

[1] Miczek, F. and Röpke, F. K. and Edelmann, P. V. F.; *New numerical solver for flows at various Mach numbers*; Astronomy and Astrophysics **576** (2015) A50; [DOI: 10.1051/0004-6361/201425059]

[2] Taylor, G. I. and Green, A. E.; *Mechanism of the Production of Small Eddies from Large Ones*; Royal Society of London Proceedings Series A **158** (1937) 499–521

[3] Drikakis, D. and Grinstein, F. F. and Youngs, D; *Simulation of transition and turbulence decay in the Taylor–Green vortex*; Journal of Turbulence **8** (2007) N20

Table 1: Strong scaling tests on 1920^3 grid. The total memory requirement is 57.5 TiB.

bg_size	rpn	MPI ranks	tpp	threads	GiB/core	time (s)
4096	16	65536	4	262144	0.90	2523.07
8192	16	131072	4	524288	0.45	993.01
12288	16	196608	4	786432	0.30	739.08
16384	16	262144	4	1048576	0.22	631.75
20480	16	327680	4	1310720	0.18	505.14
24576	16	393216	4	1572864	0.15	555.13

Table 2: Strong scaling tests on 2688^3 grid. The total memory requirement is 145.8 TiB.

bg_size	rpn	MPI ranks	tpp	threads	GiB/core	time (s)
14336	16	229376	4	917504	0.60	1472.58
21504	16	344064	4	1376256	0.40	1179.25
28672	16	458752	4	1835008	0.30	835.56

Table 3: Timings and I/O bandwidth reached when writing 264 GiB of output. Output was either written using MPI-IO to a single file (*one file*) or using C stdio using one file per process (*many files*). The files were created prior to measurement. The 16 rack test was not run in *many files* mode to avoid stress on the file system.

number of racks	4	8	16
walltime in s			
one file	45.3	29.8	54.1
many files	15.9	10.7	
bandwidth in GiB/s			
one file	5.8	8.9	4.9
many files	15.6	24.7	

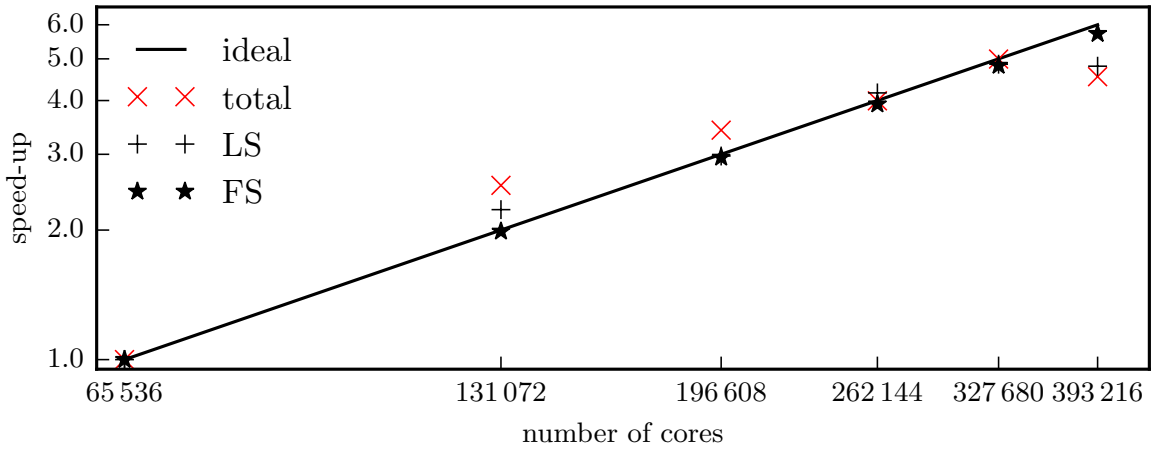


Figure 2: Strong scaling on a 1920^3 grid. The code was run with one MPI process per core and 4 OpenMP threads per process. The different markers show speed-up for the total runtime, the linear solver (LS), and the computation of fluxes and source terms (FS). The point of reference for each is the lowest number of cores. The raw data are given in Table 1.

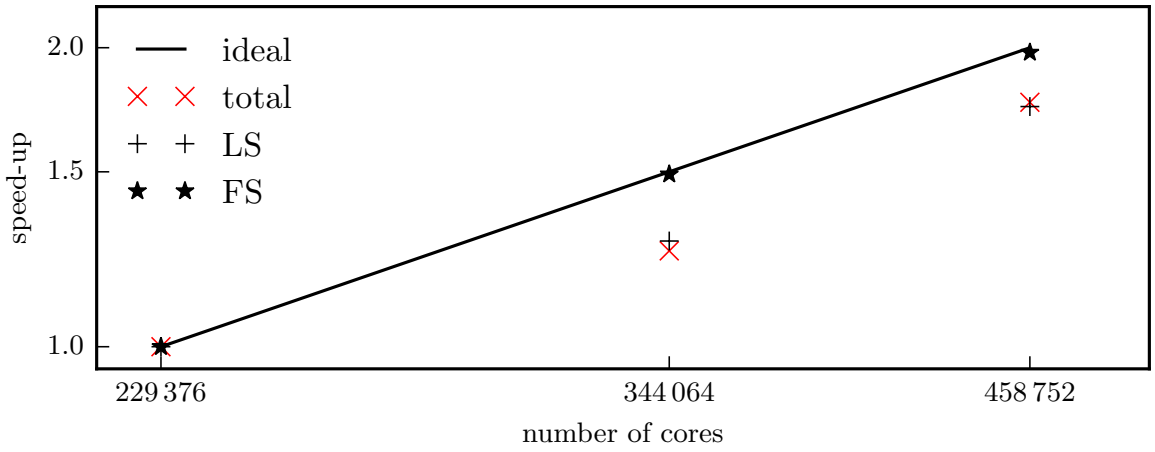


Figure 3: Strong scaling on a 2688^3 grid. The code was run with one MPI process per core and 4 OpenMP threads per process. The different markers show speed-up for the total runtime, the linear solver (LS), and the computation of fluxes and source terms (FS). The point of reference for each is the lowest number of cores. The raw data are given in Table 2.


JÜLICH
FORSCHUNGSZENTRUM